Debreceni Egyetem

Informatikai Kar

DOMináns AJAX, avagy a soha-újra-nem-töltődő webalkalmazás ideája

Témavezető:

Dr. Adamkó Attila

egyetemi tanársegéd

Készítette:

Nyáray Zoltán

PTI szakos hallgató

Debrecen

2009

Tartalom

1. Bevezető

Témánk két, a webalkalmazások készítésének menetét felgyorsító eszköz bemutatása lesz, nagyobb hangsúlyt fektetve a gyakorlati alapokra, valamint egy általunk elképzelt tartalomkezelő rendszer megvalósítására törekedve.

Megismerhetjük a jQuery JavaScript függvénykönyvtár nyújtotta lehetőségeket, látni fogjuk, hogy hogyan könnyíti meg a kliensoldali műveletek elvégzését. A jQuery szlogenje, a "write less, do more" is arra utal, hogy kevesebb programkód írásával végezzünk el minél több műveletet, ezzel gyorsítva a fejlesztési folyamatot. Mindemellett eléggé kiforrottnak tekinthető, az 1.0-ás verzió lassan három éve látott napvilágot, és olyan nevek használják webes alkalmazásaikban, mint pl. a Google, a Dell, a Bank of America, az NBC, a Mozilla a Wordpress és sokan mások.

Az Akelos PHP alapú keretrendszer Bermi Ferrer projektje, aki ezzel a Rails-re épülő Ruby on Rails keretrendszer átültetését tűzte ki célul PHP nyelvre. A Ruby on Rails szlogenje, a "Web development, that doesn't hurt" megmosolyogtathat minket, és tulajdonképpen ez is a célja. Azt szeretné kifejezni, hogy a Ruby, így ennek átültetéseként az Akelos is, az átlátható, jól strukturált és szép programkódok írását, valamint fejlesztés sebességét és a produktivitást helyezi középpontba.

Ezen rendszerek megismertetése után pedig megmutatjuk, hogy hogyan néz ki az elképzelt tartalomkezelő rendszer egy folyamatának megvalósítása használatukkal.

Mindezek előtt viszont kicsit mögé nézünk az AJAX és a DOM szavaknak, hogy később nyugodtan használhassuk az ide vonatkozó kifejezéseket, megismerjük az alapokat.

Ezzel kezdődik majd a második fejezet.

A bevezetőben még megemlíteném, hogy azért választottam ezt a témát, mert alapvetően érdekel a webalkalmazások fejlesztése. Emellett több mint egy évig dolgoztam ilyen környezetben, így mondhatni egy kevés tapasztalatra is szert tettem ezen a téren. Ezen kívül, mint a web 2.0 társadalom aktív tagja, sok közösségi és tartalommegosztó rendszert használok nap mint nap, látom a hiányosságaikat és előnyeiket. Elképzeltem egy minden szempontból ideális rendszert, amelyet szerettem volna megvalósítani, és gondoltam testhezálló lenne, ha ezt a témát választanám.

2. AJAX

AJAX = Asynchronous JavaScript And XML, azaz Aszinkron JavaScript és XML. Ez egy programozási technika, amely 2005-ben vált népszerűvé, amikor a Google megalkotta a Google Suggest-et. Nem egy új programozási nyelv, nem is programozási nyelv, hanem egy új fajta felhasználási módszere már létező szabványoknak, amely segítségével jobb, kisebb, gyorsabb és a felhasználók számára barátságosabb webalkalmazásokat készíthetünk.

Az AJAX-al a kliens oldalon található JavaScript közvetlenül kommunikálhat a szerverrel a JavaScript XMLHttpRequest objektumának segítségével. Ennek használatával a JavaScript programunk adatot cserélhet a szerverrel, anélkül, hogy az oldalt újra kellene töltenünk a böngészőben.

Aszinkron adatátvitelt használ a böngésző és a webszerver között, ezzel lehetővé téve, hogy a weboldalunk, webalkalmazásunk csak kis mennyiségű információt kérjen a szervertől egész oldalak helyett.

Az AJAX a következő webes szabványokra épül:

- JavaScript
- XML
- HTML
- CSS

Ezek a szabványok jól, pontosan definiáltak, és az összes jelentős böngésző által támogatottak, így az AJAX-os webalkalmazások mondhatni böngésző- és platformfüggetlenek.

Az AJAX célja, hogy jobb webalkalmazásokat készíthessünk. A webalkalmazásoknak sok előnyük van az asztali alkalmazásokkal szemben. Sokkal nagyobb lehet a célközönségük, könnyebb őket telepíteni, a felhasználónak ezzel szinte nem is kell foglalkoznia. A fejlesztők oldaláról pedig könnyebb őket támogatni és fejleszteni. Viszont a webalkalmazások még nem olyan gazdagok funkciókban, és nem olyan felhasználóbarátak, mint a hagyományos asztali alkalmazások. Erre hivatott az AJAX, hogy ilyen szempontból javuljanak a webalkalmazások.

2.1.Az AJAX HTTP kérésekkel operál

Hagyományos módszerrel, ha a kliens valamilyen információt szeretne kérni, mondjuk egy nevet egy adatbázisból vagy egy fájlból, ami a szerveren található, vagy valamilyen információt szeretne küldeni a szervernek, akkor ahhoz szükségünk van egy HTML form-ra, aminek tartalmát az oldal elküldi a szervernek, majd ennek következtében a szerver reagál, és a böngésző egy új oldalt betöltve megjeleníti az eredményt.

Mivel a szerver minden ilyen esetben egy új oldalt ad vissza eredményül, amit a böngészőnek teljes egészében be kell töltenie, ezek a hagyományos webalkalmazások lassúak lehetnek, és kevésbé tűnhetnek barátságosnak a felhasználó számára.

Az AJAX-al a weboldalon elhelyezett JavaScript kód közvetlenül kommunikál a szerverrel egy JavaScript XMLHttpRequest objektumon keresztül. Ennek segítségével egy úgynevezett HTTP Request-et (HTTP kérést) küldhetünk a szervernek, és úgy kaphatunk választ attól, hogy az oldal közben nem töltődik újra. A felhasználó ugyanazon az oldalon marad, és észre sem veszi, hogy a háttérben a JavaScript program adatokat kér, adatokat fogad a szervertől.

Használatával a webalkalmazás fejlesztője frissítheti az oldal tartalmát a szervertől kapott adatokkal, azután, hogy az oldal már betöltődött.

Egy egyszerű példa lehet az AJAX használtára a Google Suggest, ami úgy működik, hogy amikor a felhasználó elkezdi begépelni a keresendő kifejezést a Google keresőmezőjébe, akkor az oldalon található JavaScript erre az eseményre reagálva, egy HTTP Request használatával elküldi a szervernek az aktuálisan a mezőben található karakterláncot, aminek következtében a szerver visszaküldi a kapott keresőkifejezésre tett javaslatok listáját. Eztuán pedig az oldalon található JavaScript, érzékelve, hogy megérkezett a válasz, megjeleníti a listát a keresőmező alatt.

Az XMLHttpRequest objektum támogatott az Internet Explorer 5.0+, a Safari 1.2, a Mozilla 1.0, a Firefox, az Opera 8+ és a Netscape 7 böngészőkben.

2.2. Az XMLHttpRequest objektum

A W3C ajánlása szerint az objektum támogat mindenféle szöveg alapú formátumot az információ cseréjére, beleértve az XML-t, és használható kérések küldéséhez mind HTTP, min HTTPS protokolokon keresztül.

Nézzük meg egy egyszerűsített W3C példakódon keresztül, hogy hogyan kérhetünk az XMLHttpRequest objektum segítségével egy egyszerű XML fájlt a szervertől:

```
// a később használt eseménykezelő
function handler(){
  if(this.readyState == 4 && this.status == 200){
    if(this.responseXML != null){
      // feldolgozás
    else {
      // hiba
    }
  }
  else if (this.readyState == 4 && this.status != 200){
    // rossz oldal vagy hálózati hiba
  }
}

var client = new XMLHttpRequest();
client.onreadystatechange = handler;
client.open("GET", "test.xml");
client.send();
```

A `handler()` függvény kezeli majd az `XMLHttpRequest` objektum állapotának változásait. Ha az objektum `readyState` attribútuma = 4 (4 = "loaded") és `status` attribútuma = 200 (200 = "OK"), akkor elkezdhetjük feldolgozni a visszakapott adatokat. Mivel XML fájlt kértünk, a szerver által küldött dokumentum típusa XML, így a válasz automatikusan a `responseXML` attribútumba kerül, mint már felépített DOM dokumentum objektum. Ha valóban létezik ez az objektum, elkezdhetünk vele operálni, JavaScript segítségével megjeleníthetünk elemeket az aktuális oldalon, de el is vehetünk, meg is változtathatjuk őket. Ha mégsem létezik – bár a `readyState` és a `status` szerint kaptunk választ a szervertől – akkor arra következtethetünk, hogy hibás volt az XML fájl, amit kértünk.

Ha a `readyState` ugyan 4, azaz “loaded”, viszont a status nem 200, azaz nem "OK" értékű, akkor rossz oldalt töltöttünk be, vagy hálózati hiba adódott.

A függvény megadása után létrehozunk egy `XMLHttpRequest` objektumot, megadjuk az előbbi függvényt az objektum állapotváltozásainek kezeléséhez, megadjuk a metódust az

adatok küldéséhez (“GET” vagy “POST”), valamint az URL-t, ami a szerver oldalon azonosítja a kívánt fájlt. Végül pedig elküldjük a kérést

Bár az `XMLHttpRequest` objektum létrehozása az összes modern böngészőben a leírtak szerint működik, hozzátehetjük, hogy az Internet Explorer régebbi, 5-ös és 6-os verzióiban a következő módon kellene eljárnunk, illetve a következő módon tesztelhetnénk le, hogy melyik módszert kell alkalmaznunk:

```
// modern böngészők
if (window.XMLHttpRequest) {
  client = new XMLHttpRequest();
}
// IE5 és IE6
if (window.ActiveXObject){
  client = new ActiveXObject("Microsoft.XMLHTTP");
}
```

2.3.Dinamikus oldalak esetén

A fenti példában egy egyszerű XML fájlt kértünk a szervertől, azonban URL-ként megadhatjuk bármilyen szerver oldali szkript elérési útját is, így a dinamikusan generált tartalmat visszakapva válaszként. Ebben az esetben lehet jelentősége az `open()` függvényben a használt metódus paraméternek, ugyanis ekkor a szerver oldali programnak paramétereket is küldhetünk az `XMLHttpRequest` objektum segítségével:

```
client.open("POST", "somescript.php");
var firstName = "Zoltán";
var lastName = "Nyáray";
var params = "first_name=" + fistName + "&lastName=" + lastName;
client.setRequestHeader("Content-type",
                        "application/x-www-form-urlencoded");
client.setRequestHeader("Content-length", params.length);
client.setRequestHeader("Connection", "close");
client.send(params);
```

Most "POST"-ra állítottuk a küldés metódusát, így a szerveroldalon a somescript.php a `$_POST` változóból nyerheti ki az itt küldött paramétereket. Ebben az esetben ajánlatos megfelelően beállítani a kérés fejlécét a `setRequestHeader()` metódussal. Ezután a `send()` metódusnak átadjuk az előre összeállított paramétersztringet, elküldve ezzel a kérést.

Így a szerveroldali somescript.php szkript a következő módon juthat hozzá a küldött paraméterekhez:

```
$first_name = $_POST["firstName"];
$last_name = $_POST["lastName"];
```

Majd ezután például egy adatbázisból kikeresve az adott nevet, visszaküldheti a kliensnek valamilyen tulajdonságát egyszerű szövegként, vagy valamilyen tulajdonságait XML szövegként, amit, mint azt az első példában láthattuk, ha a szerveroldali szkript helyesen beállítja a dokumentum típusát, akkor az `XMLHttpRequest` objektum össze is állít egy XML dokumentum objektumba.

Fontos, hogy ehhez szükséges, hogy szerver oldalon helyesen beállítsuk a válasz fejlécét. Ezt PHP-ban így tehetjük meg:

```
header("Content-type: text/xml; charset='utf-8'");
```

3. DOM

A DOM (Document Object Model) egy W3C (World Wide Web Consortium) ajánlás, egy platform- és nyelvfüggetlen interfész, amely lehetővé teszi a programok számára, hogy dinamikusan hozzáférjenek és módosítsák egy dokumentum tartalmát, struktúráját és stílusát.

3.1. A DOM három részre osztható

- Core DOM – a sztenderd modell bármilyen struktúrált dokumentumhoz
- XML DOM – a sztenderd modell XML dokumentumokhoz
- HTML DOM – a sztenderd modell HTML dokumentumokhoz

A DOM definiálja az összes dokumentum elem objektumait, azok attribútumait és az elérésükhöz szükséges metódusokat. Mi most a HTML DOM-al fogunk foglalkozni, ugyanis HTML oldalak szerkezetét szeretnénk módosítani webalkalmazásunk felhasználóbarátabbá tételéhez.

3.2. A HTML DOM

- Sztenderd objektum modell a HTML-hez
- Sztenderd programozási interfész a HTML-hez
- Platform- és nyelvfüggetlen
- W3C sztenderd

A HTML DOM tehát a HTML elemek objektumait, azok attribútumait, valamint a hozzáférésükhöz szükséges metódusokat definiálja. Egyszerűbben: a HTML DOM egy sztenderd arra, hogy hogyan érjünk el, módosítsunk, adjunk hozzá, vagy töröljünk HTML elemeket a dokumentumból.

A DOM-ra hivatkozva, minden a HTML dokumentumban node-nak tekintendő:

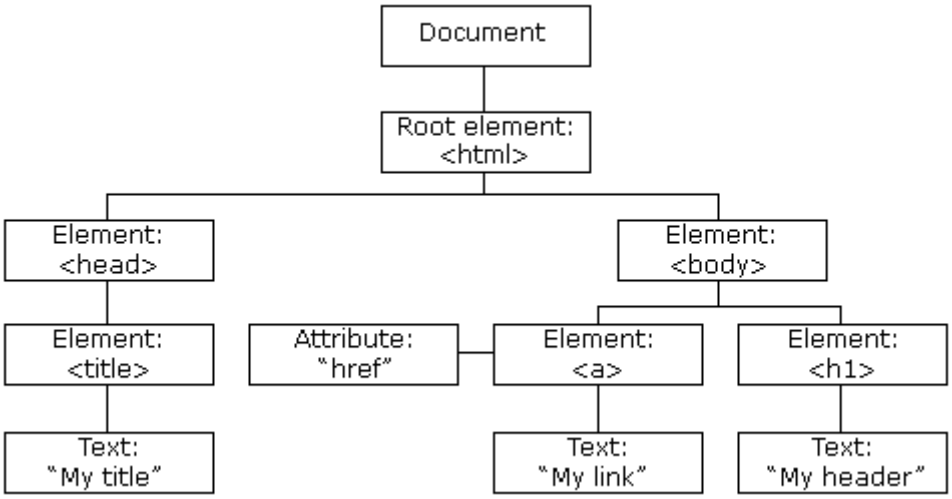
- Az egész dokumentum egy document node
- Minden HTML elem egy element node
- A HTML elemekben található szövegek text node-ok
- Minden HTML attribútum egy attribute node
- A kommentek pedig comment node-ok

3.3.Fa struktúra

A HTML DOM a HTML dokumentum node-jait egy fa-struktúrában ábrázolja, ahol a node-ok hierachikus kapcsolatban vannak egymással. Ha például adott a következő HTML dokumentum:

```
<html>
  <head>
    <title>My title</title>
  </head>
  <body>
    <a href="http://www.w3.org">My link</a>
    <h1>My header</h1>
  </body>
</html>
```

Akkor az 1. ábrán látható a dokumentumban található node-ok hierarchikus kapcsolata.



1. ábra

Ezeket a kapcsolatok lehetnek:

- Parent, azaz szülő
- Child, azaz gyermek
- Sibling, azaz testvér

Egy HTML DOM node fában:

- A legfelső node-ot root-nak, azaz gyökérnek hívjuk
- Minden node-nak, kivéve a root node-ot, pontosan egy parent node-ja van
- Egy node-nak akárhány child node-ja lehet
- Leaf, vagy levél az a node, amelyiknek nincs child node-ja
- Sibling-ek azok a node-ok, amelyeknek ugyanaz a parent node-juk

A DOM a HTML dokumentumot tehát node objektumokkal modellezi, amely node-ok elérhetőek a JavaScript nyelv használatával. A DOM programozási interfésze sztenderd attribútumok és metódusok halmazaként van definiálva.

Például, ha x egy node objektum, akkor néhány tipikus DOM attribútuma lehet:

- x.nodeName – x neve
- x.nodeValue – x értéke
- x.parentNode – x parent-je, azaz szülő node-ja
- x.childNodes – x child node-jai, azaz gyermekei
- x.attributes – x attribútum node-jai

Vagy néhány tipikus DOM metódusa lehet:

- x.appendChild(*node*) – x gyermekeként beilleszti az adott node-ot
- x.removeChild(*node*) – az adott node-ot eltávolítja x gyermekei közül
- x.getElementById(*id*) – a gyermekek közül a megadott id-jű node-ot adja
- x.getElementsByTagName(*name*) – a node adott nevű gyermekeit adja (tagname)

4. JQuery – write less, do more

A jQuery egy gyors és tömör JavaScript függvénykönyvtár, mely eszközeivel egyszerűsíti a HTML DOM-on való operálást, a node-ok kiválasztását, a közöttük való navigálást, az események kezelését és az AJAX interakciókat, mindezekkel az igen gyors fejlesztést megcélozva.

Jellemzői:

- Lightweight – kicsi, tömör
- CSS3 kész – támogatja a CSS 1-3 szelektorokat (és még többet is)
- Cross-browser – böngészők: IE 6.0+, FF 2+, Safari 3.0+, Opera 9.0+, Chrome

4.1.Szelektorok

A DOM node-ok kiválasztása jQuery-vel úgynevezett szelektorok segítségével történik, amelyeknek a hagyományos CSS szelektorok képezik az alapjait. A CSS esetében a szelektorokat arra használjuk, hogy megmondjuk, hogy például az adott tagname-el, az adott class-al vagy az adott id-vel rendelkező HTML elemeknek milyen legyen a kinézetük. A jQuery ezeket a szelektorokat használja (kicsit kibővítve) arra, hogy megtaláljon node-okat a node fában.

Vegyük például az alábbi HTML dokumentumot:

```
<html>
  <head>
    <title>My title</title>
    <style type="text/css">
      .product_desc{
        background-color: #7F7F7F;
        border: 1px solid black;
      }
    </style>
  </head>
  <body>
    <h1 class="main_header" title="Header Title">My header</h1>
    <h2 id="product_id_123">Product 123</h2>
    <p class="product_desc">
      <b>This is product 123.</b>
      <a href="http://jquery.com/" title="jQuery homepage">jQuery</a>
      <input id="input_1" value="Type text here"></input>
    </p>
    <p class="product_desc">
```

```
    </p>
    <ul id="list_1">
      <li>List item 1</li>
      <li class="selected">List item 2</li>
      <li>List item 3</li>
    </ul>
  </body>
</html>
```

Mielőtt elkezdenénk, tudnunk kell, hogy mindenfajta operációt a jQuery nevű objektum használatával végezhetünk el. Ez viszont elérhető a $ néven is, ugyanis a jQuery forráskódja így kezdődik:

```
var jQuery = window.jQuery = window.$ = function( selector, context ) {
  return new jQuery.fn.init( selector, context );
};
```

4.1.1. Context

Láthatjuk, hogy a `jQuery` objektum két paraméterrel példányosítható. Az első a `selector` nevű, itt adhatjuk meg az említett szelektort, a második pedig opcionálisan a `context` paraméter. Ez egy node objektum vagy egy jQuery objektum lehet, és azt mondja meg a jQuery objektumnak, hogy az adott szelektort milyen környezetben keresse. Ha megadjuk a context-et, akkor csak a context node-ból kiindulva keresi, azaz csak a context node leszármazottai között. Ha nem adjuk meg a context-et, akkor alapértelmezetten az egész dokumentum-ban keresi a megadott szelektornak megfelelő node-okat. Ebben a részben egyik példában sem használjuk a `context` paramétert, ennek később lesz jelentősége.

4.1.2. Kiválasztás

Tehát `jQuery` objektumot (`$`) használva egyszerűen kiválaszthatjuk a nekünk megfelelő node-okat. A legfontosabbakkal kiválaszthatunk:

CSS class alapján:

```
var productDesc = $(".product_desc");
```

Id alapján:

```
var productH2 = $("#product_id_123");
```

Tagname alapján:

```
var body = $("body");
```

Előd – leszármazott viszony és szelektorok kombinációja alapján:

```
var link = $(".product_desc a");
```

4.1.3. A jQuery keret

Az így kapott `productDesc`, `productH2` és `body` változók nem node objektumok lesznek, hanem jQuery objektumok. Ugyan DOM-ban egy node objektum biztosít metódusokat arra, hogy elérjük a gyermekeit, a szülő node-ját valamint az összes attribútumát, a DOM API sepcifikációjában és implementációiban található hiányosságok miatt ezeket a metódusokat mégsem kényelmes használni. A jQuery egy keretet biztosít a node objektumok köré, hogy segítse a DOM-al való interakciókat.

4.1.4. Halmazok

Mivel a dokumentumban több node is rendelkezhet ugyanazzal az osztállyal vagy id-vel, vagy több node-nak is lehet ugyanaz a tagname tulajdonsága, ezért bizonyos szelektorok alkalmazásával nem egy jQuery objektumot kapunk eredményül, hanem jQuery objektumok egy halmazát. Az így kapott halmazon a jQuery függvények ugyanúgy értelmezhetőek. Megjegyezhetjük, hogy a változtató metódusok általában az összes halmazbeli jQuery objektumra végrehajtják a változtatást, a lekérdező metódusok pedig általában az első halmazbeli jQuery objektumot használják. A jQuery biztosítja az eszközt ezen halmazok kezelésére, hozzáadhatunk és elvehetünk jQuery objektumokat, szelektorokkal szűkíthethjük vagy bővíthetjük a halmazt. (Erről később, a Hivatkozás részben)

4.1.5. Láncolás

Mielőtt felsorolnánk pár hasznos funkciót ebből a keretből, meg kell említenünk, hogy a legtöbb ilyen keretbeli funkció visszatérési értékként magát a jQuery objektumot adja vissza, így lehetővé téve a funkciók egymás után láncolását, és a még átláthatóbb, rövidebb forráskódok készítését:

```
// Függvények egymás után láncolása
$(".main_header").height(30).css("background-color", "#7F7F7F");
```

4.2. Attribútumok

Ezek a metódusok segítik a node-ok attribútumainak lekérdezését és beállítását.

Megjegyezzük, hogy az összes használható jQuery metódus dokumentációja (példakódokat is beleértve) megtalálható a http://docs.jquery.com oldal API Reference szekciójában. Ebben a dolgozatban nem célunk az összes ismertetése, csak a fontosabbakat emeljük ki.

Lekérdezhetjük vagy beállíthatjuk egy node akármilyen attribútumát:

```
var title = $(".main_header").attr("title");
$(".main_header").attr("title", "Modified Header Title");
```

Vagy több attribútumát egyszerre:

```
$(".main_header").attr({
  title: "jQuery docs",
  href: "http://docs.jquery.com"
});
```

Eltávolíthatjuk egy node valamilyen attribútumát:

```
$(".product_desc a").removeAttr("title");
```

Hozzáadhatunk CSS osztályokat egy node-hoz:

```
$("#product_id_123").addClass("red");
```

Ki-be kapcsolhatunk CSS osztályokat egy node-hoz. Azaz, ha a node rendelkezik az adott CSS osztállyal, akkor `removeClass()`, egyébként pedig `addClass()`:

```
$("#product_id_123").toggleClass("red");
```

Lekérdezhetjük vagy beállíthatjuk egy node HTML tartalmát:

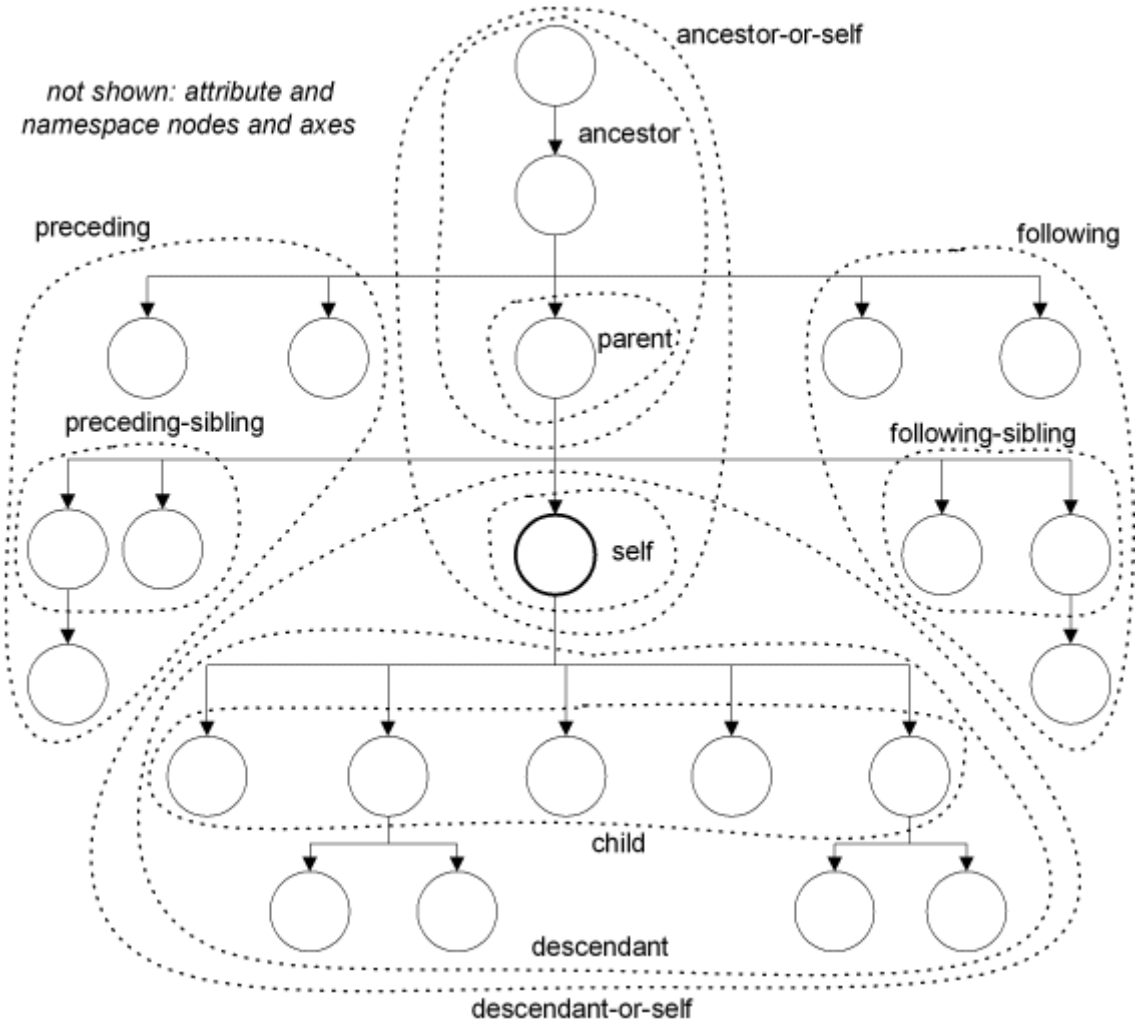
```
var inHtml = $("#product_id_123").html();
$(".product_desc").html("<h2>The content is not valid any more.</h2>");
```

Lekérdezhetjük vagy beállíthatjuk a value attribútummal rendelkező node-ok value attribútumát:

```
var inputValue = $("#input_1").val();
$("#input_1").val("Really, type text here, it's not a joke");
```

4.3.Hivatkozás (Traversing)

Ezek a metódusok segítik az egyik node-tól való eljutást valamely vele kapcsolatban álló node-hoz vagy node-okhoz. A 2. ábrán látható, hogy egy DOM node-ból milyen tengelyek mentén haladhatunk, és mely node-halmazokat hogyan nevezhetjük.



2. ábra

A jQuery segítségével, self objektumot (vagy objektumhalmazt) ismerve az összes megnevezett halmaz egyszerűen kiválasztható, közülük a legtöbb egyetlen metódus segítségével. Először viszont nézzük meg, hogy szűkíthetjük vagy bővíthetjük a talált node-ok halmazát.

Index alapján kiválaszthatunk egyet a talált node-ok halmazából. Az indexelés `0`-tól indul és `length-1`-ig tart. Az alábbi kód például csak a második `product_desc` osztályú elemhez adja hozzá a `red` osztályt:

```
var ujHalmaz = $(".product_desc").eq(1);
```

Kiszűrhetjük a talált node-ok halmazából azokat, amelyek megfelelnek a paraméterként megadott szelektornak. Ez itt például ismét csak a második `product_desc` osztályú elemet módosítja, mivel az felelt meg az `:empty` szelektornak:

```
var ujHalmaz = $(".product_desc").filter(":empty");
```

Szűrhetünk fordítva is, azaz azokat a node-okat meghagyva, amelyek nem felelnek meg az adott szelektornak:

```
var ujHalmaz = $(".product_desc").not(":empty");
```

Index alapján kiválaszthatunk egy részhalmazt a talált node-ok halmazából. Az első paraméter az első elem indexe, amelyet tartalmaznia kell az eredmény halmaznak, a második paraméter pedig az első olyan elem indexe, amelyet már nem kell tartalmaznia. A következő tehát az első két `li` node-ot tartalmazó halmazt adja vissza, és azokon operál:

```
var ujHalmaz = $("ul#list_1").slice(0, 2);
```

Hozzáadhatunk elemeket a halmazhoz szelektor alapján:

```
var ujHalmaz = $("#product_id_123").add(".main_header");
```

Kiválaszthatjuk a talált node-ok első gyermekeit (child):

```
var ujHalmaz = $("ul#list_1").children();
// Opcionálisan egy szelektort is megadhatunk paraméterként, tovább szűrve
// ezzel az eredményhalmazt:
var ujHalmaz = $("ul#list_1").children(".selected");
```

A talált node-ok halmazából elemenként kiindulva kereshetünk node-okat szelektor alapján:

```
var ujHalmaz = $("body").children().find("p");
```

Kiválaszthatjuk a talált node-ok által tartalmazott összes node-ot, rekurzívan, beleértve a text node-okat is (descendant):

```
var ujHalmaz = $("body").contents().find("a");
```

Kiválaszhatjuk a talált node-ok összes egyedi testvér (sibling) node-ját:

```
var ujHalmaz = $("ul#list_1 li.selected").siblings();
```

Kiválaszthatjuk a testvér node-ok közül az első rákövetkező node-ot, azaz a DOM fában a node mellett közvetlenül jobbra találhatót…

```
var ujHalmaz = $("ul#list_1 li.selected").next();
```

Vagy az összes rákövetkező testvér node-ot (following-siblings):

```
var ujHalmaz = $("ul#list_1 li:first").nextAll();
```

Ugyanezeket megtehetjük a megelőző node-okra vonatkozóan is a `prev()` és a `prevAll()` metódusok segítségével.

Kiválaszthatjuk a talált node-ok közvetlen szülő (parent) node-jait, vagy a node összes szülő node-ját rekurzívan, egészen a html node-ig (ancestor):

```
var ujHalmaz = $("li.selected").parent();
var ujHalmaz = $("#input_1").parents();
```

Hozzáadhatjuk az előzőleg kiválasztott node-okat az aktuálisan épp most kiválasztott node-ok halmazához (-or-self):

```
$("li.selected").parent().andSelf().css("background-color", "green");
```

4.4.Manipuláció

Ezek a metódusok segítik a DOM fa módosítását.

Lekérdezhetjük egy node kombinált szöveg tartalmát. A lekérdező `text()` metódus tulajdonképpen sorban veszi a node leszármazottjait, úgy, hogy a DOM fában az adott node gyökerű részfát inorder módon bejárja, és minden érintett node szöveg tartalmát hozzákonkatenálja az eredményhez. Ezzel tulajdonképpen lecsupaszítja a szöveget, kibontja azt a HTML környezetből. Például a példában található első bekezdés node-jának `text()` metódusa:

```
var combinedText = $("p.product_desc:first").text();
// combinedText = "This is product 123.jQuery"
```

A beállító `text()` metódus pedig majdnem ugyanazt a műveletet végzi, mint a beállító `html()` metódus, kivéve, hogy a paraméterben levédi a HTML kódot, azaz a "<" és ">" karaktereket kicseréli a nekik megfelelő HTML entitások kódjaira:

```
$("li.selected").text("Selected list item.");
```

Minden kiválasztott node belső tartalmához adja, a paraméterként kapott tartalmat. A paraméter lehet DOM node objektum, jQuery objektum vagy szöveg.

```
var toInsert = $("<li>New list item at the end.</li>");
$("#list_1").append(toInsert);
// Ezt egyszerűbben így is elvégezhetjük:
$("#list_1").append("<li>New list item 2.</li>");
```

A metódus a node-ot a végére illeszti be, azaz a beszúrt node lesz az utolsó gyermeke a szülő node-nak. Ennek a műveletnek az fordítottjaként tekinthető az `appendTo()` metódus, amely a kiválasztott node-okat illeszti be a paraméterként megkapott szelektorral kiválasztott node-ok belsejébe.

```
$("h2#product_id_123").appendTo("p:last");
```

A metódus a kiválasztott node-okat nem másolja, hanem áthelyezi. Az utóbb említett két metódushoz hasonlóan működnek a `prepend()` és a `prependTo()` metódusok, azzal a különbséggel, hogy a beszúrt node az első gyermek lesz a testvérei között.

Az `before()` és `after()` metódusokkal a talált node elé és után illeszthetünk be, azaz a beillesztett node az aktuálisan hivatkozott node testvér node-ja lesz, és közvetlenül tőle balra vagy jobbra fog elhelyezkedni a fában.

```
$("li").after("<div class='line'></div>");
$("p").before("<div class='header_image'></div>");
```

Ezek fordítottjaként tekinthetőek az `insertBefore()` és `insertAfter()` metódusok, amelyek a kiválasztott node-ot vagy node-okat a paraméterként kapott szelektorral kiválasztott node után illesztik.

```
$("#list_1").insertBefore("h1");
// Ez tulajdonképpen azonos ezzel: $("h1").before("#list_1");
$("#product_id_123").insertAfter("#list_1");
```

Becsomagolhatjuk a kiválasztott node-okat a paraméterként megadott környezetbe, amit megadhatunk HTML szövegként vagy node objektumként:

```
$("h1").wrap("<div></div>");
$("h2").wrap(document.createElement("div"));
```

Ezek a metódusok egyenként veszik a talált node-okat, és mindegyiket külön-külön csomagolják be, a `wrapAll()` metódus viszont egyben veszi a kiválasztott node-okat, és úgy csomagolja be őket.

```
$("h1, h2").wrapAll("<div class='outer'></div>");
$("h1, h2").wrapAll(document.createElement("div"));
```

A `wrapInner()` metódus az előzőekhez hasonlóan működik, azzal a különbséggel, hogy a kiválasztott node-ok gyermek node-jait (beleértve a text node-okat is) csomagolja be.

```
$("li").wrapInner("<b></b>");
$("li").wrapInner(document.createElement("b"));
```

A `replaceWith()` metódus segítségével a kiválasztott node-ot a paraméterként megadott másik node-ra cseréljük. A paraméter lehet HTML szöveg, DOM node objektum vagy jQuery objektum.

```
var originalElement = $("h2");
originalElement.replaceWith("<h3>" + originalElement.text() + "</h3>");
```

Ennek az fordítottjaként tekinthető a `replaceAll()` metódus:

```
var unordered = $("ul#list_1");
$("<ol>" + unordered.html() + "</ol>").replaceAll("ul#list_1");
```

Eltávolíthatjuk egy node összes gyermek node-ját (beleértve a text node-okat is) az `empty()` metódussal:

```
$("li.selected").empty();
```

Eltávolíthatjuk a kiválasztott node-ot a DOM fából:

```
$("li.selected").remove();
```

A jQuery 1.2.2-es verziójától kezdve ez utóbbi két metódus az eltávolított node-ok eseménykezelőit is eltávolítja.

A `clone()` metódussal lemásolhatjuk a kiválasztott node-ot. Ez hasznos lehet akkor, ha egy node másolatát szeretnénk beilleszteni valahova a fában. Opcionális paraméterként megadhatjuk, hogy a metódus az eseménykezelőket is másolja-e le, vagy sem (true vagy false):

```
$("ul#list_1 li").clone().insertAfter("ul#list_1 li:last");
// A listának így 6 eleme lesz, mivel az "li" szelektorral mind a három
// listaelemet kiválasztottuk és klónoztuk
```

4.5.CSS

Ezek a metódusok segítik a kiválasztott node-ok kinézeti tulajdonságainak lekérdezését és módosítását. A `css()` metódust három féle képpen használhatjuk az adott elem CSS tulajdonságainak lekérdezéséhez vagy módosításához.

Egy egyszerű CSS tulajdonságot megadva paraméterként visszatérési értékként az első kiválasztott elem adott CSS tulajdonsághoz tartozó értéket adja.

```
var bColor = $(".product_desc").css("background-color");
```

Két paraméterrel meghívva a metódus beállítja az összes kiválasztott node-on az első paraméterben megadott CSS tulajdonságot a második paraméterben megadott értékre:

```
$(".product_desc").css("font-weight", "bold");
```

Egy Map típusú JavaScript objektumot megadva paraméterként az összes kiválasztott node-on beállítja a map-ban található CSS tulajdonságokat a kulcs és érték párok alapján:

```
$(".product_desc").css({
  "background-color": "#ddeeff",
  "color": "#aadd11",
  "border-width": "3px"
});
```

A CSS tulajdonságokhoz tartozó kulcsokat nem feltétlenül szükséges idézőjelek közé raknunk, azonban akkor camel case írásmódot kell alkalmaznunk. Pl. `"background-color"` helyett írhatunk `backgroundColor`-t. Ez abból adódik, hogy ilyenkor a CSS tulajdonságra úgy hivatkozunk, mint a JavaScript DOM által definiált `style` objektum egy tulajdonságára, amik viszont a CSS tulajdonságoknak vannak megfeleltetve CamelCase formában írva.

Tulajdonképpen ezzel az egy metódussal bármilyen módon szabályozhatjuk egy node kinézetét. Ami gondot jelenthet még, az a node-hoz tartozó elem pozíciójának és különböző magasságainak és szélességeinek meghatározása illetve változtatása. Ezekre a műveletekre ugyan szolgáltatnak különböző metódusokat a böngészők JavaScript motorjai, viszont ezek eléggé eltérőek, és eléggé eltérő módon viselkedhetnek. Erre próbál egységes megoldást kínálni a jQuery a következő metódusokkal:

Az `offset()` metódus az első kiválasztott elem pozícióját adja meg a dokumentum-hoz viszonyítva. Visszatérési értéke egy objektum, melynek `left` és `top` tulajdonságaiban találjuk beállítva az elem bal felső sarkának pozícióját pixelekben értve.

```
var position = $(".product_desc:first").offset();
var leftOffset = position.left;
var topOffset = position.top;
```

A `position()` metódus teljesen hasonlóan működik, azzal a különbséggel, hogy pozíciót nem a dokumentumhoz, hanem az elem szülő elemének pozíciójához viszonyítva adja meg.

A `scrollTop()` és `scrollLeft()` metódusokkal lekérdezhetjük vagy beállíthatjuk, hogy egy elem - amennyiben a róla túllógó tartalom kezelése gördítősávok használatára van beállítva – gördítősávjai milyen pozícióban legyenek, azaz az elem tartalmából melyik rész látható, vagy legyen látható.

A `height()` és `width()` metódusokkal lekérdezhetjük vagy beállíthatjuk egy elem aktuális, kiszámolt magasságát illetve szélességét. Lekérdezéskor az eredményt pixel-ben kapjuk, a

beállító metódusnak viszont megadhatunk karakterláncot is paraméterként, amiben használhatjuk a CSS által definiált hossz mértékegységeket (`%, em, pt, in, cm, mm, ex vagy pc`). Amennyiben számot adunk meg paraméterként, automatikusan pixel-ben érti a megadott értéket.

Az `innerWidth()` és `innerHeight()` lekérdező metódusok az első kiválasztott elem belső szélességét és magasságát adják vissza, amibe beleértendő az elem padding vastagsága, viszont nem értendő bele az elem border vastagsága.

Az `outerWidth()` és `outerHeight()` metódusok pedig az első kiválasztott elem külső szélességét adják vissza, beleértve az elem border vastagságát. Opcionális paraméterként megadható, hogy a metódus beleszámolja-e az elem margin vastagságát is (`true` vagy `false`, alapértelmezetten `false`)

4.6.Események (Events)

Ezek a metódusok segítik az események beállítását az elemekhez. A jQuery eseménykezelő rendszere a W3C sztenderdnek megfelelően normalizálja az esemény objektumot. Az esemény objektum garantáltan átadódik az eseménykezelőnek, nem szükséges a `window.event` ellenőrzése. Az eredeti esemény objektum legtöbb tulajdonsága átmásolódik a jQuery esemény objektumába. Ez az objektum tulajdonképpen becsomagolja az eredeti esemény objektumot.

Nem megyünk bele részletesen az események kezelésébe, csak a későbbiekben használatos, az elemek eseményeit beállító, fontosabb metódusokat nézzük meg.

Speciális eseménynek számít a DOM készenlétét jelző, ami akkor váltódik ki, amikor a DOM betöltődött és készen áll, hogy rajta műveleteket hajtsunk végre. Erre az eseményre adhatjuk meg az eseményt kezelő függvényt a `ready()` metódussal, aminek egyetlen paramétere az eseményt kezelő függvény. Megadhatunk nevesített függvényt, de JavaScript-ben használhatunk névtelen függvényeket is erre a célra:

```
// Névtelen függvénnyel megadva az eseménykezelőt
$(document).ready(function(){
   alert("The DOM is loaded and ready to manipulate on it or anything.");
   $(".product_desc").css("background-color", "red");
});

// Nevesített függvénnyel megadva az eseménykezelőt
function handler(){
  alert("The DOM is loaded");
}
$(document).ready(handler);
```

Többször is meghívhatjuk a metódust, azaz több eseménykezelőt is beállíthatunk az eseményhez, amik a beállítás sorrendjének megfelelő sorrendben fognak végrehajtódni.

A `bind()` metódussal kapcsolhatunk a kiválasztott elemek mindegyikének bizonyos eseményeihez eseménykezelőt. Az első paraméterben adjuk meg, hogy mely eseményekhez szeretnénk az eseménykezelőt kapcsolni. Több eseményt is megadhatunk szóközökkel elválasztva.

A lehetséges eseménynevek: `blur`, `focus`, `load`, `resize`, `scroll`, `unload`, `beforeunload`, `click`, `dblclick`, `mousedown`, `mouseup`, `mousemove`, `mouseover`, `mouseout`, `mouseenter`, `mouseleave`, `change`, `select`, `submit`, `keydown`, `keypress`, `keyup`, `error`.

Opcionálisan a második parméterben adhatunk meg további adatokat az eseménykezelőnek amit az `event.data` tulajdonságból érhet el.

Az utolsó paraméter lesz maga az eseményt kezelő függvény, melynek - ha adtunk meg formális paramétert – átadódik az esemény objektum, amiből elérhetjük az esemény körülményeire vonatkozó adatokat (például az egér pozícióját).

```
$(".main_header").bind("click", function(event){
  alert("Mouse was at position (" + event.pageX +
                                ", " + event.pageY + ")");
});

// A this az aktuális környezetet hivatkozza, azaz azt a node objektumot
// amelyik éppen kiváltotta az eseményt…
$(".main_header").bind("dblclick", function(){
  alert("Double-click happened in " + this.tagName);
});

// Így ezt a node objektumot becsomagolva a jQuery objektumba,
// használhatjuk rá a jQuery objektum metódusait
$(".main_header").bind("mouseenter mouseleave", function(e){
  $(this).toggleClass("mouse_over");
});
```

A `one()` metódus teljesen hasonlóan működik, azzal a különbséggel, hogy az itt megadott eseménykezelő csak az esemény első bekövetkeztekor hajtódik végre, utána úgymond leiratkozik az eseményről.

A `trigger()` metódussal manuálisan is kiválthatunk egy eseményt a kiválasztott node-okon. A metódus első paramétereként adhatjuk meg, hogy mely eseményt szeretnénk kiválatani. Ezt megtehetjük úgy, hogy karakterláncként egyszerűen megadjuk az esemény nevét, a fent

felsoroltak közül, de átadhatunk egy `jQuery.Event` objektumot is. A jQuery 1.3-as verziójától kezdve egy így kiváltott esemény buborékként felúszik a DOM fában, azaz az összes szülő elemre kiváltódik, viszont ha akarjuk megakadályozhatjuk ezt az esemény objektum `stopPropagation()` metódusával, vagy az eseménykezelő false értékkel való visszatérésével.

Az `unbind()` metódus segítségével pedig leválaszthatjuk az eseménykezelőket a kiválasztott elemekről. Paraméter nélkül meghívva az összes esemény összes hozzárendelt eseménykezelője leválasztódik, első paraméterként opcionálisan megadva az eseményt csak az adott esemény kezelői választódnak le, és végül második paraméterként opcionálisan megadhatjuk csak azt az egy eseménykezelőt, amit le szeretnénk választani.

Ezekkel a funkciókkal tulajdonképpen megoldható az események beállítása az elemekre, viszont a jQuery ad néhány helper metódust is bizonyos eseményekre. Ezekről általánosan elmondható, hogy nevük az általuk kezelt esemény nevére utal, és hogy paraméter nélkül meghívva kiváltja az adott eseményt, egy függvényt megadva paraméterként pedig beállítja az adott esemény egy eseménykezelőjét a kiválasztott elemekre.

Például egy ilyen metódus a `click()`, amely az egérkattintás-ra vonatkozik:

```
// Paraméter nélkül kiváltja az eseményt, mintha rákattintottunk volna a
címre
$(".main_header").click();

// Függvényt megadva pedig beállítja az eseménykezelőt
$(".main_header").click(function(){
  alert("You clicked on the main header.");
});
```

Ezek után a többi ilyen helper metódust csak felsoroljuk:

`blur(), change(), dblclick(), error(), focus(), keydown(), keypress(), keyup(), load(), mousedown(), mouseenter(), mouseleave(), mousemove(), mouseout(), mouseover(), mouseup(), resize(), scroll(), select(), submit(), unload()`

4.7.jQuery AJAX

Most nézzük meg, hogy egyszerűsíti le a jQuery a HTTP kérések küldését és feldolgozását.

Az `ajax()` metódus betölt egy távoli oldalt HTTP kérést használva, azaz létrehoz egy `XmlHttpRequest` objektumot, a paraméterek alapján beállítja annak tulajdonságait, elküldi a kérést, majd kezeli a választ a szintén a paraméterek között megadott függvényeket használva.

A metódus azt az `XmlHttpRequest` objektumot adja vissza, amelyet létrehoz a kéréshez, viszont nekünk ezen többnyire nem kell operálnunk.

Egyetlen paramétere egy kulcs/érték párokat tartalmazó JavaScript objektum. Nézzük meg ezeket a kulcs/érték párokat:

`async` – `Boolean (alapértelmezett: true)`

Ha értéke `true`, akkor a kérés aszinkron lesz elküldve, azaz a kliensoldali kód tovább fut a kérés elküldése után, és nem vár a válaszra a szerver felől. Szinkron küldéshez állítsuk `false`-ra.

`beforeSend` – `Függvény`

Itt állíthatunk be egy függvényt, ami a kérés elküldése előtt fog lefutni. A függvény első paramétereként átadódik majd a használt `XmlHttpRequest` objektum, így ebben a függvényben lehetőségünk van azt még saját kezűleg módosítani a küldés előtt. Ha a függvényben `false`-al térünk vissza, akkor azzal megszakítjuk a kérés folyamatát, nem lesz elküldve a szerver felé.

`cache` – `Boolean (alapértelmezett: true)`

Ha értéke `false`, akkor a kért oldal nem lesz cache-elve a böngésző által.

`complete` – `Függvény`

Ez tulajdonképpen egy AJAX eseményként fogható fel, ami akkor váltódik ki, amikor a kérés befejeződött. Ekkor az itt megadott függvény hívódik meg. A `complete` esemény a `success` és az `error` események után váltódik ki, azaz az azokat kezelő függvények hamarabb hívódnak majd meg, mint ez. Két paraméter adódik át a megadott függvénynek, az első az `XmlHttpRequest` objektum, a második pedig egy sztring, ami a kérés sikerességének vagy nem-sikerességének típusát írja le.

`contentType` – `Sztring (alapértelmezett: "application/x-www-form-urlencoded")`

Adatok küldése esetén itt adhatjuk meg a tartalom-típust. Az alapértelmezett érték a legtöbb esetben megfelelő.

data – Sztring vagy Objektum

Itt adhatjuk meg a szerver felé küldendő adatokat. Ezek automatikusan sztringgé lesznek konvertálva (ha nem így adjuk meg őket), és így lesznek elküldve. Ha objektumot adunk meg, akkor annak kulcs/érték párokból kell állnia. Ha tömböt adunk meg, akkor a jQuery a következőképpen alakítja sztringgé azt:

tömb: `adatok["adat1", "adat2"]`, sztring: `"&adatok=adat1&adatok=adat2"`

dataFilter – Függvény

Ha megadjuk, ez a függvény fogja előszűrni a szervertől visszakapott adatokat. A függvénynek két paraméter fog átadódni, az első a szervertől kapott nyers válasz, a második pedig a `dataType` paraméter. A függvénynek az előszűrt adatokkal kell visszatérnie.

dataType – Sztring

Itt adhatjuk meg, hogy milyen típusú adatot várunk vissza a szervertől. Ha nem adjuk meg, akkor a jQuery a válasz MIME típusából megpróbálja kitalálni, hogy az `XmlHttpRequest responseXML` vagy a `responseText` elemét adja át a success eseményt kezelő függvényünknek. Ha viszont megadjuk a típust, akkor az alapján a jQuery átalakítja a választ, és azt adja át.

A következő típusok használhatóak:

- `"xml"` – Egy a válaszból felépített XML dokumentum objektum adódik át.
- `"html"` – A HTML sima szövegként adódik át. Meg kell jegyeznünk, hogy a beágyazott script tag-ek a DOM-ba való beillesztéskor kiértékelődnek.
- `"script"` – Kiértékeli a választ mint JavaScript szkriptet, majd átadja azt, mint sima szöveget.
- `"json"` – Kiértékeli a választ, mint JSON-t és átadja a kapott JavaScript objektumot.
- `"jsonp"` – Betölt egy JSON blokkot JSONP-t használva…
- `"text"` – Sima szövegként adja át a választ.

error – Függvény

Itt adhatunk meg egy függvényt, ami akkor hívódik meg, ha a kérés hibába ütközik. Első paraméterként átadódik neki a használt `XmlHttpRequest` objektum, második paraméterként egy sztringet fog megkapni, ami a hiba típusát írja le, harmadik paraméterként pedig opcionálisan átadódik a függvénynek egy kivétel objektum, ha kiváltódott valamilyen kivétel. A második paraméter lehetséges értékei: `"timeout"`, `"error"`, `"notmodified"` és `"parseerror"`.

global – Boolean (alapértelmezett: true)

Az itt megadott érték szabályozza, hogy a globális AJAX eseménykezelők, az `ajaxStart` és az `ajaxStop` meghívódjanak-e ezen kérés hatására.

ifModified – Boolean (alapértelmezett: false)

Ha `true`-ra állítjuk ezt az értéket, akkor csak akkor minősül majd sikeresnek a kérés, ha a válasz változott a legutolsó kérés óta. Ezt a Last-Modified fejléc ellenőrzésével végzi.

password – Sztring

Itt adhatjuk meg a jelszót a HTTP kérés autentikációjához.

processData – Boolean (alapértelmezett: true)

Alapértelmezetten a `data` kulcsnál megadott adatot a jQuery sztringgé alakítja, és úgy küldi el a szervernek, így az beleillik az alapértelmezett tartalom típusba (`"application/x-www-form-urlencoded"`). Ha DOM dokumentumot vagy más nem átalakítandó adatot szeretnénk küldeni, akkor állítsuk ezt az értéket `false`-ra.

scriptCharset – Sztring

`"jsonp"` és a `"script"` dataType és GET típus esetén kényszeríti, hogy a kérés egy megadott karakterkészlet alapján legyen fordítva. Csak akkor szükséges, ha a lokális és a távoli karakterkészlet különbözik.

success – Függvény

Itt adhatjuk meg azt a függvényt, ami akkor hívódik meg, amikor a kérés sikeresnek minősül. Két paraméter adódik át neki, az első a szervertől visszakapott adat, ami a `dataType` kulcsnál beállított típusú, a második pedig egy sztring, ami a státuszt írja le.

timeout – Szám

Beállítja a kérés lokális timeout paraméterét milliszekundumokban. Ez felülírja a globális timeout értéket, ha az be van állítva az `$.ajaxSetup`-al.

type – Sztring (alapértelmezett: "GET")

Itt adhatjuk meg a kérés típusát (`"GET"` vagy `"POST"`).

url – Sztring

Itt adhatjuk meg a kérés URL-jét.

username – Sztring

Itt adhatjuk meg a HTTP kérés autentikációjához szükséges felhasználónevet.

xhr – Függvény

Itt adhatunk meg egy függvényt az `XmlHttpRequest` objektum készítéséhez, ha a saját implementánciónkat szeretnénk használni.

Ezzel a metódussal tehát egyszerűen testreszabhatunk, elküldhetünk és feldolgozhatunk egy HTTP kérést:

```
$.ajax({
  type: "POST",
  url: "req.php",
  data: {property_1: "value_1", property_2: "value_2"},
  success: function(message){
    alert(message);
  }
});
```

5. Akelos

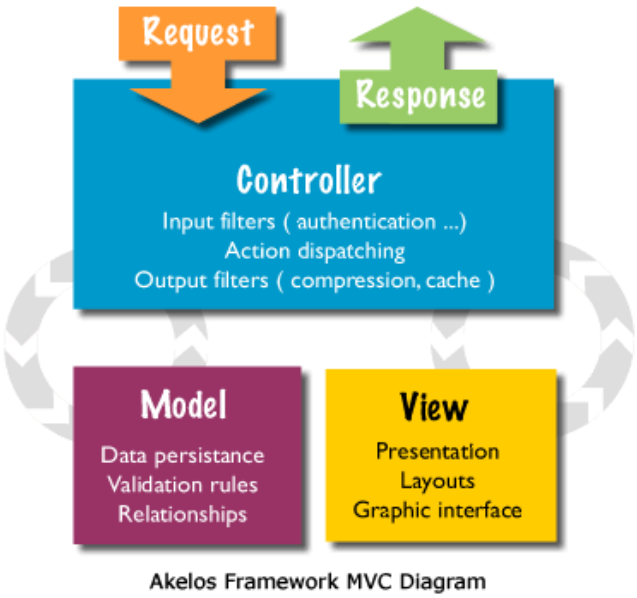
Az Akelos egy PHP nyelvre épülő keretrendszer, amely tulajdonképpen a Ruby on Rails webalkalmazás keretrendszer átültetésének tekinthető. Mint a Rails, az Akelos is az adatbázis vezérelt webalkalmazások fejlesztésének egyszerűsítését és hatékonyabbá tételét tűzi ki célul.

Lehetővé teszi a programozónak, hogy webalkalmazásokat készítsen kevesebb programkód írásával, mint más hasonló keretrendszerek esetén. Tartalmaz egy kóg generáló rendszert az alkalmazás vázának és alapjainak elkészítéséhez, nagymértékben felgyorsítva a CRUD (Create Read Update Delete) rendszer elkészítését.

5.1.Jellemzők

5.1.1. MVC

A keretrendszerben írt alkalmazások a Model View Controller tervezési mintát követik, azaz az osztályokat vagy kódrészleteket három csoportba sorolhatjuk. Az Akelos MVC diagrammja szerint a modell osztályok felelősek az adat-perzisztenciáért, végrehajtják az adatok validációját és kezelik a közöttük lévő kapcsolatokat. A view rétegben zajlik mindenféle megjelenítés layout-ok és template fájlok használatával. A controller osztályok fogadják a kéréseket, a kérés alapján elindítják a különböző action-öket, a modell osztályokon keresztül operálnak az adatokon, a view rétegnek előfeldolgozzák azokat, majd az általuk megjelenített tartalmat visszaküldik válaszként.



3. ábra

Az Akelos alapú webalkalmazások futnak a legtöbb olcsó webszerveren, mivel csak a PHP-t követeli meg, és nem igényel semmilyen nem-sztenderdnek számító PHP konfigurációt.

Megtartja a legtöbb eredeti Rails interfészt, így azok dokumentációinak nagyrész használható az Akelos-hoz is. Az Akelos (a Rails-hez hasonlóan) a Convention Over Configuration és a Don't Repeat Yourself filozófiákat vallja.

5.1.2. Konvenció a Konfiguráció Felett (Convention Over Configuration)

A programozónak csak azokat az összefüggéseket kell deklarálnia, amik nem konvencionálisak az alkalmazásban, mivel a keretrendszer számos összefüggést automatikusan felépít egyszerű elnevezési konvenciók alapján. Például, ha van egy `Product` modell osztályunk, akkor az adatbázisunkban található `products` nevű tábla ehhez a modellhez lesz kapcsolva. Itt a konvenció az, hogy a tábla neve a modell osztály nevének (angol) többes számú alakja.

5.1.3. Ne Ismételd Önmagad (Don't Repeat Yourself)

Ez azt jelenti, hogy az információknak egy helyen kell lenniük, megelőzve ezzel kétértelműséget. Az Akelos komponensek úgy vannak integrálva, hogy nem szükséges hidakat kiépítenünk közöttük. Nem szükséges például a modell osztályokban deklarálnunk a hivatkozott tábla oszlopainak megfelelő tagokat, mivel ezek automatikusan meghatározódnak az adatbázis tábla oszlopának definícióiból.

5.1.4. Aktív Rekord (Active Record) minta

Az Akelos az adatok kezeléséhez az Aktív Rekord mintát használja. Martin Fowler nevezi meg a Patterns of Enterprise Application Architecture című könyvében:

"Egy objektum ami egy adatbázis tábla vagy nézet egy sorát csomagolja be, bezárja az adatbázis hozzáférést, és tárgyköri logikát tesz erre az adatra."

Az aktív rekord köti össze az üzleti objektumokat az adatbázis táblákkal, egy tárgyköri modellt megalkotva ezzel, ahol a logika és az adat egy egységen belül vannak. Tulajdonképpen az adatbázis tábla vagy nézet egy osztálynak van megfeleltetve, az osztály egy példánya pedig egy táblabeli sort jelképez.

Az objektum hordozza az adatot és a viselkedést is. Ezen adatok nagyrésze állandó, és az adatbázisban kell tárolni őket. Az adatok eléréséhez szükséges logika magába az objektumba való behelyezésével az aktív rekord a legkézenfekvőbb megközelítést vallja. Így mindenki tudja, hogyan írják az adatbázisba és hogyan olvassák az adatbázisból az adatokat.

5.1.5. Sintags

Az Akelos beépítetten támogat egy Sintags nevű template nyelvet, ami nagyban segíti egyszerűbb nézetek gyors megírását. Ez egy limitált egyközkészlettel bíró template nyelv, ami nem a PHP nyelv használatának kiváltására szolgál, csupán néhány egyszerű szerkezet egyszerűsítésére, rövidítésére.

5.1.6. Elnevezési konvencióik

A táblákat angol nyelv szerinti többes számban nevezzük el, a szavakat alulvonás karakterek választják el egymástól, mint például: `steering_wheels`.

Az Akelos magától kezeli a létrehozási és a módosítási dátum és idő adatokat, így ezeket nekünk nem kell leprogramoznunk.

Ha egy mező nevét `"_at"`-re végződően adjuk meg, akkor az akelos automatikusan beállítja a mező típusát `datetime` típusra, ha `"_on"`-ra, akkor pedig date típusra. Például a `posted_at` mező alapértelmezettként `datetime`, a `posted_on` pedig `date` típusú lenne.

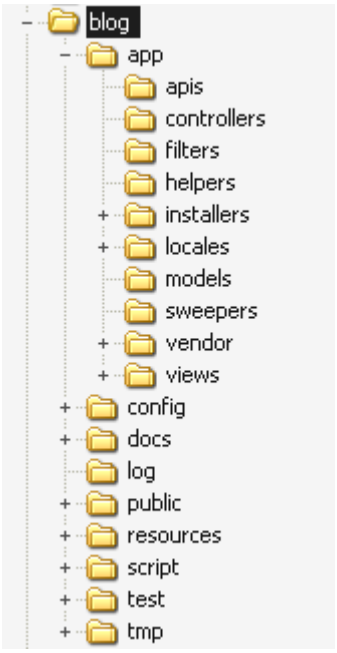
A modelleket egyes számban nevezzük el, az első betű nagybetű, ezután pedig CamelCase írásmódot használunk, mint például: `SteeringWheel`.

5.2. Telepítés

A http://www.akelos.org/download címről letöltve és kicsomagolva a keretrendszert, egyszerűen elkezdhetjük annak telepítését. A script könyvtárban találhatóak az Akelos számunkra igen hasznos szkriptjei, amik tulajdonképpen PHP programok. Nem kell mást tennünk, mint futtatnunk ezek közül a `setup` nevű szkriptet, ami létrehozza projektünket a megadott helyre (ehhez persze szükségünk van PHP értelmezőre, és annak nem árt a PATH-ban lennie). Ha például AppServ szerverünk a kicsomagolt Akelos keretrendszerrel egy könyvtárban található, akkor a szerverünk www könyvtárába egyszerűen létrehozhatjuk blog nevű projektünket a következő paranccsal:

```
php .\script\setup ..\AppServ\www\blog
```

Ekkor a következő könyvtárszerkezet készül el:



A számunkra leglényegesebb könyvtárak az **app** könyvtáron belül találhatóak:

- **controllers** – Ide kerülnek majd a controller osztályok.
- **helpers** – Ide kerülnek a view-okban és a controller-ekben használható segítő osztályok.
- **installers** – Itt lesznek a migráláshoz szükséges osztályok.
- **locales** – Ide kerülnek a lokalizáció szótárfájljai.
- **models** – Itt lesznek a model osztályok.
- **views** – Ide pedig a view fájlok, a template-ek kerülnek.

Az alkalmazásunk további fontosabb könyvtárai pedig:

- **config** – Konfigurációs fájlok.
- **public** – Ide tehetjük a publikus fájlokat, mint például a JavaScript fájlok, CSS fájlok, képek és egyéb média. *NIX rendszerek esetén automatikusan létrejön egy softlink, ami ide mutat, Windows alatt viszont meg kell adnunk egy alias-t erre a httpd.conf fájlban, valahogy így:

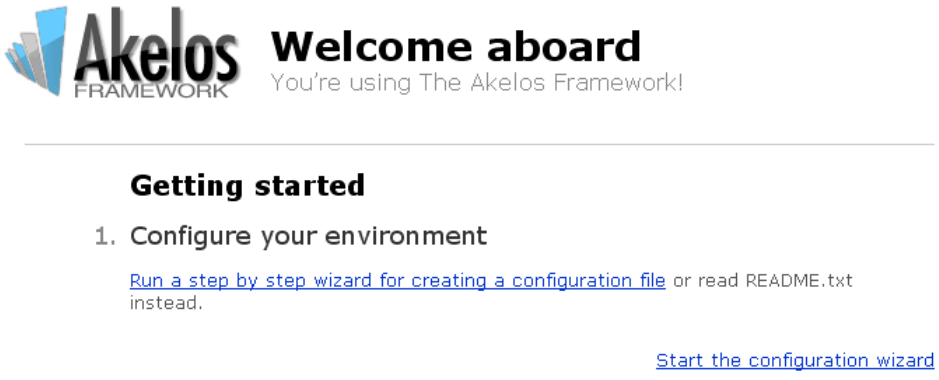
```
Alias /blog "/path/to_my/blog/public"

<Directory "/path/to_my/blog/public">
    Options Indexes FollowSymLinks
    AllowOverride All
    Order allow,deny
    Allow from all
</Directory>
```

- **script** – itt találhatóak az Akelos parancssorból indítható szkriptjei
- **tmp** – itt pedig az Akelos által használt cache, és a template-ekből összeillesztett, csak PHP kódot tartalmazó view fájlok

Az alkalmazásunk ekkor egy alapértelmezett keretrendszer telepítő alkalmazás osztályait tartalmazza, így a böngésző címsorába begépelve a projektünk elérési útját a 4. ábrán látható képpel találjuk szembe magunkat.

http://localhost/blog:

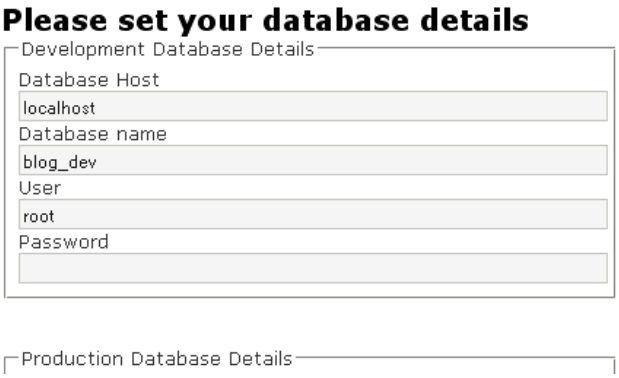


4. ábra

Ezután a Start the configuration wizard hivatkozásra kattintva a keretrendszert telepítő alkalmazásunk megkeresi, hogy milyen adatbázisokat támogat a jelenlegi szerver konfigurációnk. Jelen esetben az AppServ-ünk MySQL adatbázist támogat, így csak ezt kínálja fel. Miután kiválasztottuk, az alkalmazás három adatbázis adatait kéri tőlünk. Ez azért

van, mert Akelos-nak 3 különböző futási környezete van, és mindegyiknek külön adatbázisa. (Ezeket az adatbázisokat létre kell hoznunk.) A development módban történik az alkalmazás fejlesztése, testing módban teszteljük, és production módban adjuk ki a nagyközönség számára. Erősen megkövetelik, hogy a következő konvenciók alapján nevezzük el ezeket az adatbázisokat:

- `adatbazis_nev_dev` – a development módhoz (fejlesztés, ez az alapértelmezett mód)
- `adatbazis_nev` – a production módhoz (kiadás)
- `adatbazis_nev_tests` – a testing módhoz (tesztelés)



5. ábra

Ha helyesen adtuk meg az adatokat, és léteztek ezek az adatbázisok, akkor a telepítő alkalmazás elmenti az adatokat a konfigurációs fájlokba, és nekünk ezzel többet nem kell foglalkoznunk.

Ezután a telepítő megkérdezi, hogy milyen nyelveken szeretnénk majd elérhetővé tenni a készítendő alkalmazásunkat. A nyelvi szótárfájlok ezen beállítás alapján fognak kigenerálódni.

6. ábra

Ha ezt beállítottuk, egy **Welcome aboard** üzenet jelzi, hogy a telepítés véget ért, tulajdonképpen innentől kezdődik az alkalmazásunk fejlesztése.

5.3. Models

5.3.1. Generátorok

Ezután elkezdhetjük a modell osztályaink generálását. Az Akelos parancssorból indítható szkriptjei közül a generate nevűvel tudunk kódot generálni. Ha paraméter nélkül hívjuk meg, megkapjuk az elérhető generátorok listáját:

```
php .\script\generate

You must supply a valid generator as the first command.
Available generator are:
service
scaffold
model
mailer
controller
clone
```

5.3.2. Modell generátor

Először generáljuk le a modelljeinket. Adatvezérelt alkalmazásunknál a tervezés tulajdonképpen itt kezdődik, el kell döntenünk, hogy milyen szerkezete lesz az adatbázisunknak, milyen táblák és milyen kapcsolatok legyenek. A modelleket az adatbázis tábláknak feleltetjük meg, ezek azok az osztályok, amik közvetlenül az adatokon operálnak. Mi a következő modelleket hozzuk most létre:

- Post – Egy post a blogunkban.
- CTxt – Egy szöveg konténer.
- Txt – Egy szöveg.
- PostCTxt – A Post és a CTxt modellek kapcsolódási pontja.
- CTxtTxt – A CTxt és a Txt modellek kapcsolódási pontja.

Az előbb felsorolt generátorok közül a model nevűt használva tehát a Post modell fájljait az alábbi módon hozhatjuk létre:

```
php .\script\generate model Post
```

A parancs létrehozza a post.php fájlt a models könyvtárban, a post_installer.php fájlt az installers könyvtárban (valamint a teszteléshez szükséges állományokat a test könyvtárban, amikkel most nem foglalkozunk).

post.php:

```
class Post extends ActiveRecord { }
```

post_installer.php:

```
class PostInstaller extends AkInstaller{
  function up_1(){
    $this->createTable("posts", "id, name");
  }

  function down_1(){
    $this->dropTable("posts");
  }
}
```

5.3.3. Installer-ek

Láthatjuk, hogy a modell osztályunk az ActiveRecord osztályt terjeszti ki, valamint, hogy az installer osztályba bekerült két metódus. Az Akelos az agilis fejlesztést előtérbe helyezve kínál megoldást az adatbázis szerkezetének létrehozásához és későbbi módosításához. Ezek az ún. installer-ek és a beépített migrációs rendszer, amellyel könnyedén lépkedhetünk az adatbázis különböző verziói között. Ez az installer osztály lesz az adatbázis verziók kezelésének alapja.

A `PostInstaller` osztály a `posts` tábláért felelős. Az `up_1()` metódusba kerül a táblát létrehozó utasítás, a `down_1()` metódusba pedig a táblát eltávolító. Ez követően, ha módosítani szeretnénk valamit ezen a táblán, akkor az `up_2()` metódusba írjuk a táblát módosító utasítást, a `down_2()` metódusba pedig az ezen változtatásokat visszavonó utasításokat, a módosító utasítások iverzeit (inverz sorrendben). Így válik lehetővé, hogy oda-vissza lépkedjünk a különböző verziók között. Az adatbázis módosításához számos metódust kínál az Akelos, amiket megtalálhatunk az `AkInstaller` osztály dokumentációjában.

Az általunk tervezett posts tábla installer-e így néz ki:

```
class PostInstaller extends AkInstaller{
  function up_1(){
    $this->createTable("posts",
      "id,
       title,
       unique_css text,
       posted_at");
  }
```

```
  function down_1(){
    $this->dropTable("posts");
  }
}
```

Az Akelos-nak ennyi elég ahhoz, hogy létrehozza a táblánkat. Az `id` mező lesz az elsődleges kulcs, a `title` mező az alapértelmezett varchar típusú lesz, a `unique_css` mezőnek mi adtuk meg a típust, ez `text` típusú lesz, a `posted_at` mező pedig – mivel `"_at"`-re végződik – az Akelos szerint automatikusan `datetime` típusú lesz.

A mezők típusainak megadásakor használhatjuk a következőket:

```
integer (vagy int)
float
datetime, date, timestamp, time
text
string
binary
boolean (vagy bool)
```

Vagy használhatunk bármilyen oszloptípust, amelyet az adatbázisunk támogat, például `varchar`-t vagy `smallint`-et MySQL esetén.

5.3.4. Migráció

Miután megírtuk az installer osztályunkat, az Akelos `migrate` szkriptjével hajthatjuk végre a változatásokat az adatbázison. Ennek első paramétereként megadjuk, hogy melyik installer osztályt akarjuk használni. Második paraméterként az `install` vagy `uninstall` szavak egyikét. Az install opcióval frissítjük az adatbázist, azaz magasabb verzióra, felfelé migrálunk, az uninstall-al pedig visszavonjuk a változtatásokat, azaz kisebb verzióra, lefelé migrálunk. Végül harmadik paraméterként opcionálisan megadhatjuk, hogy mely verzióig tartson a migrálás (ha nem adjuk meg, akkor a felfelé a legmagasabb, lefelé pedig a legkisebb, nullás verzióig végzi el a változtatásokat). A mi esetünkben:

```
php .\script\migrate post install
```

A szkript kiírja az elvégzett módosításokat a képernyőre:

```
(mysqlt): CREATE TABLE posts (
id              INTEGER NOT NULL AUTO_INCREMENT,
title           VARCHAR(255),
unique_css      LONGTEXT,
posted_at       DATETIME,
updated_at      DATETIME,
created_at      DATETIME,
```

```
        PRIMARY KEY (id)
)TYPE=InnoDB
```

Mint láthatjuk az `updated_at` és a `created_at` mezőket az Akelos automatikusan hozta létre, és ezek után automatikusan is fogja frissíteni ezeket, nekünk erre nem kell figyelnünk.

Hozzuk létre ugyanígy a `CTxt` modellünket is, generáljuk le a modellt a következő paranccsal:

```
php .\script\generate model CTxt
```

Írjuk meg az installer osztályt (installers/c_txt_installer.php):

```
class CTxtInstaller extends AkInstaller{
  function up_1(){
    $this->createTable("c_txts",
      "id,
       title,
       unique_css text,
       posted_at");
  }

  function down_1(){
    $this->dropTable("c_txts");
  }
}
```

Majd migráljuk az adatbázist a következő paranccsal:

```
php .\script\migrate c_txt install
```

5.3.5. Több-a-többhöz kapcsolat

A `posts` és a `c_txts` táblák között több a többhöz kapcsolatot szeretnénk létrehozni, azaz egy Post-hoz több CTxt tartozhat, és egy CTxt több Post-hoz is tartozhat. Ezt a kapcsolatot egy kapcsolótáblával, és egy hozzá tartozó kapcsolómodellel hozhatjuk létre.

Generáljunk egy modellt `PostCTxt` néven, ez lesz a kapcsolat modellje:

```
php .\script\generate model PostCTxt
```

Írjuk meg az installer osztályt így:

```
class PostCTxtInstaller extends AkInstaller{
  function up_1(){
    $this->createTable("post_c_txts",
      "id,
       post_id,
```

```
        c_txt_id");
  }

  function down_1(){
    $this->dropTable("post_c_txts");
  }
}
```

Majd migráljuk az adatbázist:

```
php .\script\migrate post_c_txt install
```

Az Akelos az "_id" végződésű mezőkről tudja, hogy külső kulcsként szeretnénk őket alkalmazni, így be is állítja ezeket automatikusan, mint azt láthatjuk is a kimeneten:

```
(mysqlt): ALTER TABLE post_c_txts ADD  INDEX idx_post_c_txts_post_id
(post_id)
(mysqlt): ALTER TABLE post_c_txts ADD  INDEX idx_post_c_txts_c_txt_id
(c_txt_id)
```

A kapcsolódás alapja tehát adatbázis szinten már készen van, most nézzük meg, milyen módosításokat kell végeznünk a modelleken.

A modell osztályokban az Akelosnak egyszerű változók beállításával megadhatjuk, hogy mely modell mely más modellekkel van kapcsolatban, és milyen kapcsolat van közöttük. Ezek a változók a következők:

$has_many – a modell egy egy-a-többhöz kapcsolat bal (egy) oldalán áll

$belongs_to – a modell egy egy-a-többhöz kapcsolat jobb (több) oldalán áll

Azaz, ha lenne egy Author (szerző) modellünk, és egy Post-hoz egy Author tartozhatna, egy Author-hoz pedig több Post, akkor így adnánk meg ezt a kapcsolatot:

models/author.php:

```
class Author extends ActiveRecord{
  $has_many = array("posts");

}
```

models/post.php:

```
class Post extends ActiveRecord{
  $belongs_to = array("author");

}
```

Megfigyelhetjük, hogy a kapcsolódó modellek megadásánál kisbetűs írásmódot alkalmaztunk (a szavakat alulvonással választanánk el), valamint, hogy a `$has_many` esetén a modellek nevei többes számban, a `$belgons_to` esetén pedig egyes számban vannak.

Megadhatunk ezenkívül néhány tulajdonságot a kapcsolatra vonatkozóan úgy, hogy egy asszociatív tömb kulcsai jelentik a kapcsolódó modellt, az értékei pedig a beállításokat, szintén egy asszociatív tömbben, például így:

```
class Author extends ActiveRecord{
  $has_many = array("posts" => array("dependent" => "destroy",
                                     "order" => "posted_at"));

}
```

Két beállítást adtunk meg az előző kapcsolatunkra. Az első a `dependent`, itt `"destroy"`, ami azt jelenti, hogy minden, ezen Author objektumunkhoz kapcsolt Post objektum semmisüljön meg ezzel az objektummal együtt, úgy, hogy hívódjon meg azok `destroy()` metódusa. A második az `order`, ami meghatározza, hogy az Author objektumunkhoz kapcsolt Post objektumok milyen sorrend alapján kerüljenek hozzánk az adatbázisból. Ebben az esetben a `posted_at` mező alapján, növekvő sorrendben.

Ez a helyzet egy egy-a-többhöz kapcsolat esetén. Mi viszont több-a-többhöz kapcsolatot szeretnénk kialakítani a Post és a CTxt, valamin a CTxt és a Txt modellek között. A kapcsolótáblákat és azok modelljeit már létrehoztuk, most állítsuk meg a kapcsolatot a modellekben (csak a CTxt és a Txt kapcsolatot nézzük meg):

models/c_txt.php:

```
class CTxt extends ActiveRecord{
  var $has_many = array("c_txt_txts" => array("dependent" => "destroy",
                                              "order" => "number"));

}
```

models/txt.php

```
class Txt extends ActiveRecord{
  var $has_many = array("c_txt_txts" => array("dependent" => "destroy"));

}
```

models/c_txt_txts.php

```
class Txt extends ActiveRecord{
  var $belongs_to = array("c_txt", "txt");

}
```

A kapcsolatot a CTxt oldalról szeretnénk támogatni egy sorszámmal, azaz a CTxt-ken belül a Txt-ket valamilyen sorrendben szeretnénk majd megjeleníteni. Ehhez a `c_txt_txts` kapcsolótáblában elhelyezünk egy number mezőt, azaz az installer osztályt így írtuk meg:

```
class CTxtTxtInstaller extends AkInstaller{
  function up_1(){
    $this->createTable("c_txt_txts",
      "id,
       c_txt_id,
       txt_id,
       number integer");
  }

  function down_1(){
    $this->dropTable("c_txt_txts");
  }
}
```

Ezután a CTxt modellben hozzunk létre egy metódust, amely az aktuális CTxt objektumhoz betölti a hozzá kapcsolódó Txt-ket. Azt szeretnénk, hogy a CTxt modellünk `txts` nevű tagjában legyenek a kapcsolódó Txt objektumok. Ehhez deklarájuk a `txts` nevű tagot, valamint írunk egy metódust, amely összegyűjti ezeket a kapcsolótáblán keresztül:

models/c_txt.php:

```
class CTxt extends ActiveRecord{
  var $has_many = array("c_txt_txts" => array("dependent" => "destroy",
                                              "order" => "number"));

  public $txts;

  // Ez lesz a kapcsolódó Txt-ket összegyűjtő metódus
  function loadTxts(){
    if($this->c_txt_txt){
      foreach($this->c_txt_txt->load() as $ctt){
        $this->txts[] = $ctt->txt->load();
      }
    }
  }
}
```

Mivel az Akelos a controller osztályokban (később) automatikusan betölti a kapcsolótábla ide tartozó sorait, így nekünk már csak az ezekhez tartozó Txt-ket kell betöltenünk, és hozzáfűznűnk a gyűjtőhöz.

Így egyszer meghívva a CTxt objektum `loadTxts()` metódusát bárhol elérhetjük a kapcsolódó Txt objektumokat a `txts` tagon keresztül.

5.4.Controllers

Miután létrehoztuk a hiányzó modelleket és felkészítettük őket a kapcsolatok kezelésére, a modell résszel egy időre készen is vagyunk. A kérések kiszolgálását viszont a controller osztályok végzik, amik még láthatóan nincsenek készen. A controllers könyvtárban egyedül a page_controller.php fájl található, ezzel a tartalommal:

```
class PageController extends ApplicationController{
  function index(){ }
}
```

Láthatjuk, hogy az osztálynak van egy `index()` metódusa. Ezeket a metódusokat hívjuk action-öknek. Általánosan elmondható, hogy minden egyes megjeleníteni kívánt oldalhoz egy ilyen action tartozik. A kívánt action megjelenítéséhez az böngészőben írjuk az URL végére a következőt:

```
/controller_nev/action_nev
```

Például így:

```
http://localhost/blog/post/listing
```

Ebből az Akelos tudni fogja, hogy a `PostController` osztály `listing()` metódusát kell meghívnia. A controller a modell osztályok használatával betölti a szükséges adatokat, elvégzi a szükséges műveleteket, majd átadja az magát egy view-nak, ami megjeleníti a tartalmat (MVC).

5.4.1. URL Rewrite

Az, hogy url-ben ez ilyen világosan látszik, az az URL Rewrite technikának köszönhető. Ehhez szükséges, hogy az Apache szerverünkön be legyen töltve ez a modul

Apache/conf/httpd.conf:

```
LoadModule rewrite_module modules/mod_rewrite.so
```

Valamint, hogy az alkalmazásunk gyökérkönyvtárában található .htaccess fájlban bekapcsoljuk a modult, és definiálva legyenek az újraírás szabályai. Ezeket az Akelos már megtette helyettünk:

blog/.htaccess:

```
RewriteEngine on
RewriteBase /blog
RewriteRule ^(.*)$ index.php?ak=$1 [L,QSA]
```

Láthatjuk, hogy a szabály bal oldalán található reguláris kifejezés az URL "/blog" utáni részét (a RewriteBase miatt) teljesen megeszi, és átírja az URL-t, úgy, hogy az index.php-nek egy `"ak"` nevű paraméterben átadja a végét. Ez az index.php fogja betölteni a keretrendszert, és hívja meg a megfelelő controller megfelelő metódusát.

Az URL Rewrite technika több szempontból is igen hasznos lehet.

- A felhasználó számára könnyebb olvashatóságot biztosít, az URL alapján is meg tudja határozni, hogy éppen melyik részén van az oldalnak, valamint könnyebben meg is tudja jegyezni azt.
- Előnyös lehet keresőoptimalizálás szempontból is. Ha például egy cikk címe közvetlenül megjelenik az URL-ben, akkor a kereső emiatt előrébb fogja helyezni a találati listában.
- Az URL-ek nem nyújtanak információt az oldalunk belső működésével kapcsolatban. A dinamikusan generált oldalak URL-jében a paraméterek ugyanis alapot nyújthatnak egy támadónak.
- Egy bizonyos oldal URL-je megmaradhat akkor is változatlanul, ha a háttértechnológia megváltozik.

Visszatérve az `index()` action-höz, ha nem mondunk mást egy action-ben, akkor a benne található utasítások után az Akelos automatikusan betölti az views/controller_nev/action_nev.tpl fájlt, amely előállítja a kimenetet, és végül ez jelenik meg a böngészőnkben.

A controller-eket és a view-okat legenerálhatjuk az osztályainkhoz a következő szkripttel:

```
php .\script\generate controller Post show listing edit
```

A controller generátor vár egy controller nevet, ami lehet CamelCase vagy alulvonással elválasztott írásmódú, ezután pedig opcionálisan megadhatunk neki action neveket. Ekkor a szkript a következőket generálja:

- controllers\post_controller.php – amiben üres metódusokként megjelennek a megadott action nevek.
- helpers\post_helper.php – ami egy teljesen üres helper osztályt tartalmaz
- views\post\show.tpl, listing.tpl, edit.tpl – amik üresek
- Valamint teszteléshez szükséges fájlokat a test könyvtárban.

5.4.2. Scaffolding

Scaffolding-nak nevezzük, amikor az adatvezérelt alkalmazásunk alapjaként létrehozzuk a CRUD (Create Read Update Delete) műveletekhez szükséges részt. A scaffold tulajdonképpen ez a része az alkalmazásnak. Az Akelos képes automatikusan scaffold-ot generálni a generate szkripttel így:

```
php .\script\generate scaffold Post
```

A generátor vár egy modell nevet, ami után opcionálisan megadhatunk egy controller nevet, és opcionális extra action neveket. Ha nem adjuk meg a controller nevet, akkor automatikusan a modell neve lesz a controller neve is. A generátor a következőket hozza létre:

- Ha nem létezik a modell, akkor a modellt és az installer-t.
- A controller-t a következő action-ökkel: `index, listing, show, add, create, edit, update, destroy` (plusz az opcionálisan megadott extra action nevek).
- Az action-ökhöz tartozó view-okat, azaz az `index, listing, show, add, create, edit, update, destroy` és az opcionálisan megadott action nevekhez tartozó template fájlokat.
- A Post-hoz tartozó layout fájlt (később).
- Valamint a teszteléshez szükséges fájlokat.

A létrehozott action-ök mind teljesen működőképesek, azaz ha a modellünk már létezett, és az adatbázisunk is készen állt, akkor az így kapott scaffold-al máris az alkalmazásunk segítségével módosíthatjuk az adatokat:

http://localhost/blog/post/listing:

Tasks:

Create new Post

Posts

Title	Unique css	Posted at	Updated at	Created at			
teszt post	font-weight:bold;	2009-05-08 14:21:00	2009-05-08 14:23:11	2009-05-08 14:21:43	Show	Edit	Delete
teszt post 2	body{ background-color:black; }	2009-05-08 14:22:00		2009-05-08 14:22:29	Show	Edit	Delete

7. ábra

Ha akarnánk, ezeken az alapokon akár el is kezdhetnénk felépíteni az alkalmazást, viszont a mi igényeinknek ez kevésbé felel meg, mivel a módosítást AJAX használatával fogjuk végezni.

5.5. Views

A view fájlok felelnek a megjelenítésért. Ezek a views/controller_nev könyvtárban található template (.tpl) fájlok. Álatalánosan elmondhatjuk, hogy minden, amit a böngészőben látni fogunk, az a template fájlok által megjelenített formában kerül oda. Ezekben a fájlokban – csakúgy, mint a HTML fájlokban – használhatunk HTML-t, és használhatunk beágyazott PHP szkripteket. Ezen felül használhatjuk a korábban említett Sintags template nyelvet is.

5.5.1. Layout-ok

Hogyan zajlik ez a megjelenítés pontosan?

Ezek a view template-ek önmagukban nem tartalmazzák az egész megjelenítendő oldalt, csak a tartalom részt szolgáltatják. Ez elsősorban tervezési megfontolásból van így. Egyrészt képzeljük csak el, milyen lenne, ha minden view template-et valahogy így kellene kezdenünk:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
```

Aztán fel kellene sorolnunk mindegyikben a stílusleíró lapokat, valamint a használt JavaScript forrásokat. Másrészt mi történne akkor, ha módosítanunk kellene mondjuk a HTML DTD-t (mindegyikben módosítanunk kellene). Ezt a keretet tehát kiemeljük a layout template-ekbe, amikben elhelyezünk a view templatek-nek egy behelyettesítési pontot, valahogy így:

```
<body>
  {content_for_layout}
</body>
```

Ezek lesznek tehát a layout fájlok, amik az app/views/layouts könyvtárban vannak, nevük megegyezik a controller nevével (kisbetűs, alulvonással elválasztott írásmódban), kiterjesztésük pedig ugyanaz, mint a view template-eknek, azaz .tpl.

Mint már említettük, a controller, miután végzett a műveletekkel, automatikusan meghívja az aktuális action-nel megegyező nevű template-et. Most már azt is elmondhatjuk, hogy ennek a template-nek a tartalmát az alapértelmezett layout-ba, azaz a controller nevével megegyezőbe rakja bele.

Ha módosítani akarjuk ezt az eljárást, akkor azt megtehetjük a controller osztály `render()` metódusával, amelyet az `AkActionController` ősosztályától örököl. Természetesen, ha saját kezűleg jelenítjük meg a tartalmat, akkor nem hívódik meg utána az alapértelmezett template. A `render()` metódus igen sokoldalú, megjeleníthetünk vele:

- Más action-ökhöz tartozó template-et, az aktuális layout-tal, layout nélkül vagy tetszőleges layout-tal.

- Partial, azaz részleges template-eket. A részleges template-ek különösen hasznosak lehetnek, ha az oldalak gyakran ismétlődő elemekből állnak, mivel ilyenkor ezeket részleges template-ekbe kiemelve egy helyen módosíthatjuk azokat, megőrizve a konzisztenciát.
- Tetszőleges template-et, megadva a template elérési útját a .tpl kiterjesztés nélkül, relatívan a template gyökér-hez (app\views). Ilyenkor az aktuális layout kerül alkalmazásra.
- Tetszőleges fájlt, fájlrendszer elérési utat megadva. Ilyenkor üres layout-ra került a tartalom.
- Szöveget. Alapértelmezetten az aktuális layout-tal, tetszőleges layout-tal vagy layout nélkül.
- Inline template-et, ha a template dinamikusan adott. Például, ha azt szeretnénk, hogy a felhasználó egyéni template-eket készítsen, és ilyenkor letároljuk ezeket az adatbázisban.
- Vagy semmit. Például egy AJAX hívás esetén, ami csak változtat, nincs szüksége semmilyen válaszra, vagy például akkor, ha csak egy státusz kódot szeretnénk küldeni.

A `render()` metódus és számos ide vonatkozó metódus dokumentációja megtekinthető az Akelos API dokumentáció AkActionController.php részében.

5.5.2. Helper-ek

A helper-ek tulajdonképpen a helper osztályokban található metódusok. Ezek a legkisebb önálló részegységei a megjelenítésnek. Gyakran használt, apró dokumentum részletek megjelenítéséhez használjuk őket, mint például egy gyakran használt link vagy gomb, egy HTML tag, vagy egy WSYIWYG szerkesztő váza. Ha sorba akarnánk rendezni a megjelenítéshez használandó eszközeinket, aszerint, hogy hogyan épülnek egymásba, akkor az valahogy így nézne ki:

- Helper – Kisebb dokumentum rész.
- Partial template – Nagyobb dokumentum rész.
- View template – Action szint.
- Layout – Controller szint.

Természetesen csak rajtunk múlik, hogy milyen apró részekre daraboljuk a megjelenítést, hogy mit emelünk ki helper-ként, vagy hogy mihez használunk partial template-et. A válaszok mindenesetre erősen függnek az alkalmazás nagyságától.

Meg kell említeni, hogy nem csak template-ekben használhatunk helper-eket, hanem controller-ekben és más helper-ekben is.

Egy nagyon egyszerű példa lehet az automatikusan generált scaffold egy helper-e, amely a show action-höz tartozó linket jeleníti meg:

```
function link_to_show(&$record){
  return $this->_controller->url_helper->link_to(
    $this->_controller->t("Show"),
    array('action' => "show", "id" => $record->getId())
  );
}
```

A `$record` paraméterben kapja meg a modell objektumot, amelyet meg szeretnénk jeleníteni, ha rákattint a felhasználó erre a linkre. Aztán meghívja a `link_to()` URL helper-t. A link megjelenítendő szövege a `"Show"` lesz, méghozzá a `t()` metódus által többnyelvűsítve, a hivatkozás pedig az aktuális controller `show` action-jére fog mutatni a `$record`-ból kapott `id`-vel.

Ezek után egy template-ben egy ilyen show hivatkozás nekünk ennyibe fog kerülni:

```
// Ha a post_helper.php-ben található az előbbi helper
<?= $post_helper->link_to_show($post) ?>
```

6. Az alkalmazás

6.1. Célok

Célunk hasonlóvá tenni a szerkesztési, (admin) és a mindenki által látható felületet. Azaz azt szeretnénk, hogy miközben a felhasználó szerkeszti az általa megosztott tartalmat, mindig egyből hasonló képet lásson, mint amilyet az oldalra tévedve látni fognak a látogatók.

Ezen felül a lehető legtöbb módosítást szeretnénk AJAX használatával, az oldal újratöltése nélkül megoldani. Természetesen csak akkor, ha ez előnyösebb, mint a hagyományos megoldás. Ugyanis, ha például olyan felületre navigál a felhasználó, ahol majdnem az egész oldalt ki kellene cserélnünk, akkor hatékonyabb lehet az oldal újratöltése.

8. ábra

A szerkesztési felületen többek között minden szöveg úgynevezett in-place-editor-okkal lesz szerkeszthető. Ez azt jelenti, hogy a szöveg úgy jelenik meg, ahogy azt az oldalra látogatók látni fogják, rákattintva viszont kicserélődik egy beviteli mezőre, amelyben már szerkeszthető.

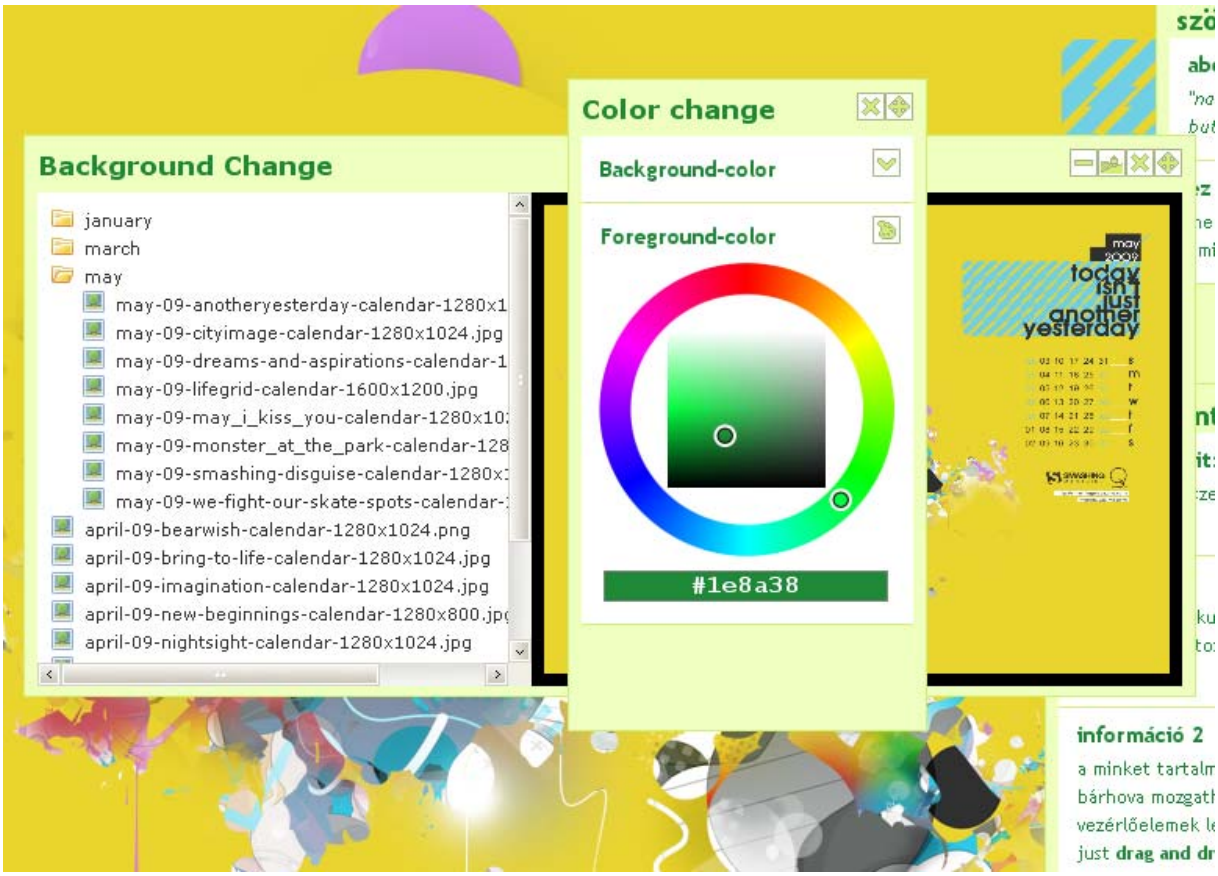
Ezután ha a beviteli mező elveszíti a fókuszt, vagy lenyomjuk az enter billentyűt, AJAX-al elmentésre kerül az új érték. A stílusok megfelelő beállításával azt is célul tűzzük ki, hogy szerkesztés közben is ugyanúgy nézzen ki, mint nem-szerkesztési módban.

A felületen található szöveg konténerek pozícióját szeretnénk letárolni adatbázisban, majd a szerkesztési felületen drag-and-drop módszerrel szeretnénk módosítani helyzetét. A drag-and-drop művelet végén az AJAX kérés automatikusan elmenti az új pozíciót, és felvillanással jelzi a mentés sikerességét.

A szöveg konténerekhez egy gomnyomással adhatunk hozzá szöveg elemeket, és ugyanígy módosíthatjuk azok sorrendjét is. Láthattuk, hogy a modell tervezésekor sorszámot rendeltünk ehhez a kapcsolathoz.

Bármely elem törlése vagy eltávolítása is egyetlen gombnyomással, az oldal újratöltése nélkül elvégezhető.

A Post és a CTxt elemek CSS tulajdonságait ugyanígy, AJAX segítségével szeretnénk módosítani egy univerzális, CSS szerkesztő felületen, amelyet a nem szakavatott felhasználók is képesek használni.



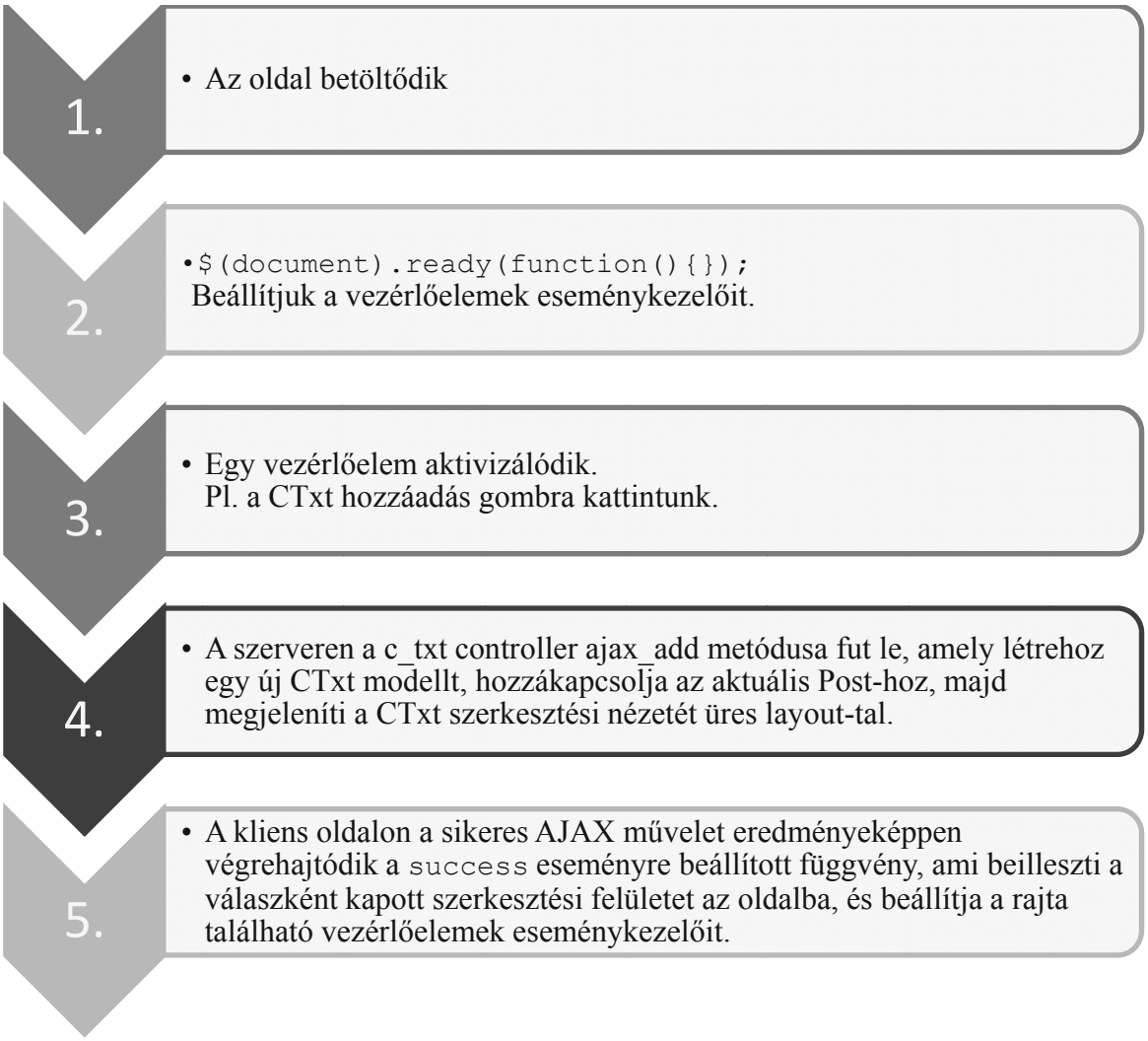
9. ábra

A felsoroltak közül egyedül ez utóbbi nincs még benne az alkalmazásban, a többi pont mind megvalósításra került.

Ezen felül elkészült a színszerkesztő felület, amellyel bármely szín értékű CSS tulajdonságot módosíthatjuk majd, valamint egy kép böngésző felület, amelyet most a Post hátterének módosítására használunk.

6.2.Egy folyamat

Nem célunk az egész alkalmazás készítésének bemutatása, mivel az meghaladná e dokumentum terjedelmi korlátait, viszont az alábbi példafolyamat nagyszerűen bemutatja, hogyan is képzelhetőek el ezek a műveletek.



6.3. Az oldal betöltődik

Az oldal összeállítása során figyelünk arra, hogy a vezérlőelemek megjelenítésekor helyesen állítsuk be a CSS osztályt, a vezérelt művelet URL-jét, valamint az egyéb küldendő adatokat. Az URL-t és az esetleges paramétereket a vezérlő HTML elem egy-egy attribútumaként adjuk meg. Például a CTxt hozzáadása gomb HTML kódja így néz ki:

```
<div class="control_button add_control_button" element_id="post_1"
parent_id="1" url="/blog/c_txt/ajax_add/"></div>
```

Az első CSS osztály csak a kinézet miatt van, a második, az `add_control_button` osztály alapján állítódik majd be az eseménykezelő. Az `url` attribútum tárolja a végrehajtandó AJAX kérés URL-jét. Az `element_id` attribútum azt mondja meg, hogy melyik HTML elembe kell majd beilleszteni az új CTxt elemet, a `parent_id` pedig azt, hogy a controller-nek melyik Post-hoz kell hozzákapcsolnia azt az adatbázisban. Ezen paraméterek megadásával tulajdonképpen egységesítettük a hozzáadó vezérlőelemek eseménykezelőit, ugyanis az ezen attribútumok alapján (URL és paraméterek) fogja beállítani az elvégzendő AJAX művelet URL-jét és a küldendő adatokat.

A szerveroldalon a view template-ben ennek a vezérlőelemnek a megjelenítését egy helper segítségével tesszük, így:

```
$wysiwyg_helper->control_button(array(
  "type" => "add_control_button",
  "element_id" => "post_" . $post->id,
  "url" => array(
    "controller" => "c_txt",
    "action" => "ajax_add"),
  "element" => array(
    "parent_id" => $post->id)
));
```

A `control_button` helper a wysiwyg_helper.php fájlban, a `WysiwygHelper` osztály egy metódusa, amelyet minden controllerben elérhetővé tettünk, úgy, hogy a controller osztályok ősosztályában, az `ApplicationController`-ben ezt mondtuk:

```
var $app_helpers = "wysiwyg";
```

Ebbe a változóba felsorolva (vesszővel elválasztva), az itt megadott helperek globálisan láthatóak lesznek.

Végül a `control_button` helper:

```
function control_button($options){
  if(!isset($options["element"]["class"]))
    $options["element"]["class"] = "control_button " . $options["type"];
  else
    $options["element"]["class"] .= " control_button " . $options["type"];

  if(isset($options["url"]))
    $options["element"]["url"] =
      $this->_controller->url_helper->url_for($options["url"]);

  if(isset($options["element_id"]))
    $options["element"]["element_id"] = $options["element_id"];

  $button_element = $this->_controller->tag_helper->content_tag(
    "div",
    "",
    $options["element"]
  );
  return $button_element;
}
```

Láthatjuk, hogy az URL attribútumot a controller-t és action-t megadó asszociatív tömbből az `url_for` helper készíti, magát az elemet pedig a `content_tag` helper állítja össze.

Mivel helper-t írtunk hozzá, így tulajdonképpen bármilyen vezérlőgomb megjeleníthető a template-eken a `control_button` helper megfelelő paraméterezésével.

6.4. Eseménykezelők beállítása

Ennyi történt szerver oldalon. Ezután az oldal betöltődik, és a vezérlőelemek eseménykezelőit CSS class alapján beállítják az inicializáló JavaScript függvények, amelyek a DOM készenlétét jelző esemény bekövetkeztekor futnak le. A hozzáadó vezérlőelemek inicializáló függvénye így néz ki:

```
function initAddControlButtons(elementObject){
  $(".add_control_button", elementObject).click(function(){
    var controlElement = $(this);
    $.ajax({
      type: "POST",
      url: $(controlElement).attr("url"),
      data: {parent_id: $(this).attr("parent_id")},
      cache: false,
      success: function(message){
```

```
        var insertElement = $(message);
        $("#"+$(controlElement).attr("element_id")).append(insertElement);

        initRemoveControlButtons(insertElement);
        initAddControlButtons(insertElement);
        initMoveAndSave(insertElement);
        initDeleteControlButtons(insertElement);
        initDownControlButtons(insertElement);
        initUpControlButtons(insertElement);
        initSingleLineEdits(insertElement);
        initMultiLineEdits(insertElement);
      }
    });
  });
}
```

Láthatjuk, hogy a függvény az paraméterként kapott `elementObject` node-on belül kiválasztja az összes `add_control_button` osztállyal rendelkező elemet, és a `click` eseményre beállít hozzájuk egy eseménykezelőt. A függvényt optimalizálási szempontból paramétereztük fel így, ezt pár sorral lentebb kifejtjük. Kezdetben, az oldal betöltődésekor ezt a `document` paraméterrel hívjuk meg, azaz a dokumentum összes `add_control_button` osztályú elemére beállítja az eseménykezelőt.

Ez az eseménykezelő tulajdonképpen egy AJAX kérés küldése lesz. Előtte létrehozunk egy változót, ami az ezen vezérlőelemünk jQuery objektumát jelenti majd, mivel az `ajax` függvényen belüle nem hivatkozhatunk már rá, mint `$(this)`.

Az AJAX kérés URL-jét és a küldendő adatot (`parent_id`) az aktuális vezérlőelem attribútumaiból állítja be, majd megadja a sikeres művelet bekövetkeztekor végrehajtandó függvényt (`success`).

Ez a függvény a kapott válaszból összeállít egy DOM objektumot (`$(message)`), majd a vezérlőelem `element_id` attribútuma alapján, az adott id-jű elem utolsó gyermekeként beilleszti azt a node fába, majd a beállítja a most beillesztett elemen található vezérlőelemek eseménykezelőit.

Visszatérve az inicializáló függvény paraméterezésére, mostmár láthatjuk, hogy például ezt az inicializáló függvényt is meghívjuk a sikeres AJAX kérés esetén, ugyanis a beillesztett CTxt szerkesztési felületen szintén vannak hozzáadó vezérlőelemek. Ekkor nem lenne célszerű a dokumentum összes `add_control_button` osztályú elemére beállítani az eseménykezelőt, ugyanis vannak olyanok, amelyekre már megtettük ezt. Így csak az éppen beillesztett elemben keressük az ilyen osztályú vezérlőelemeket.

Ezután, ha egy ilyen hozzáadó vezérlőelem aktivizálódik, azaz rákattintunk, akkor lefut a fenti eseménykezelő, és elindul a HTTP kérés.

6.5. A szerveren

A szerveren ennek hatására lefut a `"blog/c_txt/ajax_add"` URL-hez tartozó action, ami a CTxt controller `ajax_add` metódusa lesz:

```
function ajax_add(){
  $this->c_txt = new CTxt();
  $this->c_txt->save();
  $this->c_txt->addToPost($this->params["parent_id"]);

  $this->render(array("action" => "edit", "layout" => "emptylayout"));
}
```

Az action tehát létrehoz egy új CTxt modell objektumot, elmenti azt, azaz bekerül az adatbázisba, majd hozzákapcsolja a paraméterként kapott id-jű Post-hoz.

Az `addToPost()` metódus a `CTxt` modell osztályban van, és csak annyit csinál, hogy ha nem létezik az adott kapcsolat a `post_c_txts` kapcsolótáblában, akkor létrehozza azt:

```
function addToPost($post_id){
  $connect = new PostCTxt();
  if(!$connect->findFirst("post_id = ? AND c_txt_id = ?",
                          $post_id, $this->id)){

    $connect->setAttributes(array(
      "post_id" => $post_id,
      "c_txt_id" => $this->getId()
    ));

    $connect->save();
  }
}
```

A controller ezután a `render()` metódussal megjeleníti ugyanezen controller edit action-jéhez tartozó view template-et, méghozzá üres layout-tal, és ezt a HTML kódot küldi vissza válaszként a kliensnek.

6.6.Ismét a kliensen

Mint azt az eseménykezelő beállításánál láthattuk, ekkor a sikeres AJAX kérés eredményeképpen lefut ennek az eseménynek a kezelője (success), ami megjeleníti az oldalon a CTxt szerkesztési felületét, és beállítja a rajta található vezérlőelemek eseménykezelőit, ezzel aktivizálva őket.

7. Összefoglalás

A fenti példafolyamat kifejtése után láthatjuk, hogy milyen egyszerű a megvalósítás a használt eszközökkel. Alig 200 sor pogramkódot kellett írnunk, és működik egy CTxt hozzáadása egy Post-hoz. Ezen felül a további műveletek megvalósítása a bemutatotthoz teljesen hasonlóan elvégezhető, mondhatni lineárisan megy, valamint a megírt kódok egy része újra felhasználható.

Ez köszönhető annak, hogy a DOM módosítása során a jQuery, a szerveroldali műveletek részéről pedig az Akelos rengeteg terhet levett a vállunkról. Nem kellett foglalkoznunk azzal, hogy a különböző böngészők esetleg eltérő módon valósítják meg a DOM API-t, mint ahogy azzal sem, hogy kialakítsuk a szerver oldali szkriptek megfelelő struktúráját. Láthattuk, hogy csak metódusokat írtunk a generált osztályokba, a működtetésért pedig az Akelos felelt.

Ezek az eszközök nagyban felgyorsítják a webalkalmazások készítését, így több időnk jut figyelni az olyan összetevőkre, amelyek a felhasználók szemében barátságosabbá tehetik az alkalmazást. Ilyenek például a kinézet, az ergonómia és az optimalizálás, a válaszidő csökkentése.

A projekt jövője az alkalmazás több felhasználó számára is használhatóvá tétele, minden ehhez szükséges modell és felület megalkotása, optimalizálás, és a lehető legbarátságosabb szerkesztési mód elkészítése.

Ezen dokumentum olvasásának idejekor az eddig elkészült funkciók már remélhetőleg megtekinthetőek a http://evolotion.hu/blog/post/edit/1 címen.

Irodalomjegyzék

1. Bear Bibeault, Yehuda Katz: jQuery in Action
   Manning Publications, 2007
2. Dave Crane, Darren James, Eric Pascarello: Ajax inAction
   Manning publications, 2006
3. Lee Babin: Ajax with PHP
   Apress, 2007
4. Akelos PHP Framework
   http://www.akelos.org/
5. jQuery: The Write Less, Do More JavaScript Library
   http://jquery.com/
6. Ruby on Rails
   http://rubyonrails.org/
7. W3Schools Online Web Tutorials
   http://w3schools.com/
8. World Wide Web Consortium – Web Standards
   http://www.w3.org/